O'REILLY

GitHub Copilot

Программирование в паре с ИИ

Брент Ластер

Предисловие
Эндрю Стеллмана

SPRINT book

Learning GitHub Copilot

Multiplying Your Coding Productivity Using AI

Brent Laster

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

GitHub Copilot

Программирование в паре с ИИ

Брент Ластер

SPRINT book
2026

ББК 32.973.2-018
УДК 004.413
Л26

Ластер Брент

Л26 GitHub Copilot. Программирование в паре с ИИ. — Астана: «Спринт Бук», 2026. — 304 с.: ил.

ISBN 978-601-12-5027-6

Освойте навыки программирования в тесном сотрудничестве с генеративным ИИ, используя GitHub Copilot — пожалуй, самый популярный ИИ-инструмент разработчика. Брент Ластер, опытный девелопер и преподаватель, описал реальные сценарии из ежедневной практики и тем самым показал, что GitHub Copilot — это не просто подсказки в редакторе, а полноценный партнер по программированию, который может в разы ускорить вашу работу, если знать, как с ним взаимодействовать.

Книга идеально подходит разработчикам, тестировщикам, DevOps-инженерам и всем, кто так или иначе причастен к созданию ПО, независимо от уровня опыта. Вы освоите не только базовые функции, но и более мощные возможности: Copilot Edits, Copilot Agent и даже Copilot Vision, узнаете, как создавать собственные расширения и интегрировать их в свой рабочий процесс, независимо от выбранного языка программирования.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.413

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

ISBN 978-1098164652 англ.

Authorized Russian translation of the English edition of Learning GitHub Copilot ISBN 9781098164652 © 2025 Tech Skills Transformation, LLC.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-601-12-5027-6

© Перевод на русский язык ТОО «Спринт Бук», 2026
© Издание на русском языке, оформление ТОО «Спринт Бук», 2026

Краткое содержание

Оглавление

Отзывы о книге

«GitHub Copilot. Программирование в паре с ИИ» — практическое и исчерпывающее руководство, основанное на повседневном опыте разработчиков.

Эндрю Стеллман (Andrew Stellman),
разработчик, тимлид, преподаватель и автор

Каждый программист должен владеть навыками программирования с помощью ИИ, а GitHub Copilot — очевидный лидер в этой области. Книга Брента поможет овладеть этим мощным инструментом.

Том Толли (Tom Taulli),
автор книги AI-Assisted Programming

Эта книга — недостающее звено между использованием Copilot и истинным раскрытием его потенциала. Это бесценный источник информации для разработчиков, которые хотят вывести свои рабочие процессы на новый уровень.

Баладжи Дхамодхаран (Balaji Dhamodharan),
руководитель в области Data Science

Книга поможет вам шагнуть в будущее программирования. Она научит вас пользоваться инструментом, упрощающим процесс написания, понимания кода и его оптимизации, предоставляя доступные уроки и примеры использования GitHub Copilot. Кем бы вы ни были, новичком или опытным программистом, книга поможет повысить вашу продуктивность.

Адитья Гоэл (Aditya Goel),
консультант по генеративному ИИ и помощник
вице-президента международного банка

Здесь подробно описаны возможности GitHub Copilot и приведены практические примеры. На мой взгляд, это весьма содержательная книга.

Чхая Метани (Chhaya Methani),
Microsoft

В книге фундаментальные знания мастерски сочетаются с примерами их применения, поэтому ее можно порекомендовать каждому, кто хочет изучить процессы разработки с помощью искусственного интеллекта.

Сундип Гоуд Катта (Sundeep Goud Katta),
ведущий разработчик

Посвящается Анне-Марии — соавтору замечательной книги,
которой стала наша совместная жизнь.

Предисловие

Я провел немало времени, помогая разработчикам научиться использовать инструменты на базе искусственного интеллекта (ИИ) в повседневной работе. Я обучал людей, проводил семинары и писал книги по этой теме — и при создании реальных программ сам пользовался теми же инструментами. Один из них явно выделяется на общем фоне. Я говорю о GitHub Copilot.

Он произвел настоящую революцию в программировании. Я пользуюсь им постоянно.

Я пишу код не один десяток лет и, как и каждый разработчик, иногда оказываюсь в тупике — например, когда осваиваю новую технологию либо API или когда разбираюсь с плохо написанной документацией. Или когда провожу рефакторинг особенно проблемного кода, от которого моя команда старается держаться подальше. Или когда мне просто не нравится писать однообразный шаблонный код. Его нужно написать, но это будет стоить мне полчаса жизни, которые никто не вернет.

В таких ситуациях Copilot может серьезно помочь. Он может выдать готовый и работоспособный код. А может дать только отправную точку, от которой мы совместно будем двигаться к нужному результату. В любом случае это чрезвычайно ценный инструмент, который почти всегда помогает мне выйти из тупика и вернуться к обычной работе.

Впрочем, не стоит представлять работу с Copilot как сплошной праздник — к ней нужно подходить ответственно. Copilot может писать код, который компилируется, выполняется и даже проходит тесты. Но если вы не понимаете, что вам дает этот инструмент или почему работает (или, что самое важное, как вы получили результат), то вы на самом деле не контролируете то, что попадает в вашу кодовую базу. Эта книга поможет вам вернуть контроль над происходящим.

Хорошему разработчику потребуется сделать нечто большее, чем просто изучить возможности Copilot. Нужно будет освоить эффективный метод работы с ИИ, который помогает делать то, что вы уже делаете: писать код, принимать решения и решать задачи совместно с другими людьми.

Обучение начинается с понимания того, что Copilot неидеален — и он даже не пытается быть таковым. Он пытается помочь вам решать рабочие задачи, генерируя предложения, основанные на закономерностях и контексте вашего кода. И во многих случаях у него это неплохо получается. Но, как и любой инструмент, Copilot лучше всего работает, когда вы умеете управлять им и понимаете, когда стоит притормозить и перепроверить то, что он вам дает.

Именно по этой причине так важна эта книга. «GitHub Copilot. Программирование в паре с ИИ» — практическое и хорошо продуманное руководство, которое основано на повседневной работе программистов с кодом. Брент Ластер (Brent Laster) мастерски справился со своей задачей, поскольку не ограничился простым перечислением возможностей Copilot — он показал, как использовать их для выполнения реальной работы, когда ситуация сложна и постоянно меняется, а сроки поджимают.

Книга начинается с описания сильных сторон Copilot, таких как объяснение кода и написание тестов. Далее рассказывается, как этот инструмент помогает справляться с затруднениями, с которыми сталкиваются программисты при работе с запросами SQL, регулярными выражениями, файлами YAML и при решении других мелких (а иногда и не очень!) задач, отнимающих много времени. Мне нравятся примеры из книги, поскольку они отражают разновидности задач, которые разработчики решают в реальной жизни. Думаю, они понравятся и вам.

Copilot становится важным инструментом повседневной разработки не потому, что обещает некое волшебство, а потому, что учитывает особенности мышления разработчика. Автор этой книги делает то же самое: он предполагает, что вы умны, способны и продолжаете учиться, и дает вам возможность самостоятельно принимать хорошие решения, а не просто повторять приведенные примеры.

Занимаясь обучением людей и написанием книг, я видел, насколько быстро растет уровень разработчиков, если они получают четкие рекомендации, особенно когда такие советы помогают им принимать более эффективные решения в реальной работе. Эта книга хороша тем, что в ней нет громких заявлений и чрезмерных упрощений. Это практическое руководство по использованию мощного инструмента, написанное для разработчиков, которые хотят держать свой код под контролем.

Если вы еще осваиваете Copilot, то эта книга станет отличной отправной точкой. Если у вас уже есть опыт работы с ним, вы сможете освоить приемы, помогающие увеличить полезность этого инструмента и повысить доверие к нему. Если вы скептически относитесь к применению ИИ в разработке, это тоже нормально. В книге показано, как использовать Copilot максимально эффективно, не отказываясь от того, что делает вас хорошим разработчиком.

Эндрю Стеллман,
разработчик, тимлид, преподаватель и автор
книг Learning Agile, Head First C# и др.
Бруклин, Нью-Йорк, апрель 2025 г.

Введение

> Ветер и волны всегда на стороне более умелого мореплавателя.
>
> *Эдвард Гиббон (Edward Gibbon)*

Я пишу код очень много лет (так много, что даже страшно представить) и неоднократно видел, как отрасль разработки программного обеспечения трансформировалась под влиянием перемен. Сдвиги парадигмы (такие как возникновение Интернета, CI/CD, контейнеров и облачных платформ) сломали старые подходы, привели к инновациям и появлению невероятных инструментов, а также кардинально изменили методы создания программных продуктов. Среди прочего эти перемены вызвали неуверенность и замешательство и у разработчиков — они пытались понять новшества и разобраться в том, как адаптировать свои навыки к изменившейся ситуации и применять их на практике.

Теперь появился генеративный ИИ — и вызвал, возможно, самую большую волну перемен, с которой мы сталкивались в технологической области, особенно при разработке ПО. Как и другие новшества, генеративный ИИ породил масштабные инновации и серьезные изменения. Но в каком-то смысле он больше напоминает цунами, приносящий мощные новые инструменты, модели и возможности, в то время как программисты, подхваченные волной, изо всех сил пытаются удержаться на плаву.

Я хочу помочь вам оседлать волну (по крайней мере в том, что касается использования такого мощного ИИ-инструмента, как GitHub Copilot), поэтому написал эту книгу. Не стану утверждать, что в ней есть ответы на все ваши вопросы, но хочу надеяться, что она поможет вам освоить этот инструмент, понять его возможности (от базовых до расширенных) и что благодаря этому вы сможете облегчить свою работу гораздо больше, чем когда-либо ранее.

За последние годы я использовал Copilot разными способами: в своих персональных проектах, как бывший директор DevOps, запускавший его пилотное использование в исследовательской корпорации, как преподаватель Copilot в нескольких фирмах, как скептик и как энтузиаст. Что касается двух последних ролей, то на первых порах Copilot не произвел на меня такого большого впечатления, как другие инструменты. Но за то время, пока я писал эту книгу, разрабатывал учебные курсы и наблюдал за эволюцией Copilot, он стал одним из основных инструментов, которые я регулярно использую во время работы в моей интегрированной среде разработки (integrated development environment, IDE) или на GitHub. Copilot сильно изменился даже за несколько последних месяцев. За это время Copilot заметно продвинулся вперед — и в наборе функций, и в общем удобстве использования. На страницах этой книги я постарался показать как можно больше его возможностей.

Одной из моих целей при написании глав было желание рассказать о преимуществах использования Copilot и о проблемах, которые могут при этом возникнуть, а также помочь вам избежать неожиданностей. Поэтому, прежде чем переходить к основному материалу, я перечислю некоторые важные моменты.

- За последний год или около того практически в каждом выпуске GitHub Copilot изменялись внешний вид и местонахождение по крайней мере одного диалогового окна или элемента управления. Снимки экранов и описания, приведенные в книге, актуальны на момент ее написания. Но учитывайте, что к тому времени, когда вы будете читать ее, некоторые из элементов переместятся, изменят внешний вид и т. д.
- Copilot выпускает новые функции достаточно регулярно. И со временем в нем появятся новые функции, не представленные в книге. По той же причине на момент написания этих строк часть его функций помечены как *предварительные (preview)* или *экспериментальные*. Возможно, к тому моменту, когда вы будете читать эту книгу, их уже введут в эксплуатацию или удалят, если они не пройдут отбор.
- Copilot работает во многих средах, в том числе в разных IDE. Каждая IDE имеет свои особенности в плане интеграции, оформления и даже функциональности. Так сложилось исторически, что GitHub для внедрения новой функциональности и добавления полного набора функций использовал преимущественно Visual Studio Code и среды с тем же интерфейсом (например, GitHub Codespaces). По этой причине все снимки экранов и примеры в книге сделаны в VS Code или в Codespace.
- Если вы используете не VS Code, а другую IDE, то помните, что GitHub может не реализовывать в ней некоторые функции или возможности. Актуальную информацию по этой теме можно найти в документации Copilot (https://docs.github.com/en/copilot) и/или IDE.
- В свою очередь, IDE, интегрированные с Copilot, обычно могут работать на разных платформах, таких как Windows, Mac и Linux. Рассматривать в книге все вариации было бы нереально, поэтому все примеры подготовлены на Mac. Таким образом, окна и элементы управления могут отражать визуальный стиль этой операционной системы. Кроме того, на разных платформах используются разные сочетания клавиш. Чтобы не перечислять все при каждом упоминании, я выбрал клавишу Meta для обозначения сочетаний, соответствующих вашей платформе.

Все эти моменты учитываются по всей книге, которая была написана, чтобы вы могли освоить базовый уровень работы с Copilot, а затем перейти к расширенному. Главы не обязательно читать по порядку; кроме того, вы можете пропускать некоторые из них, исходя из ваших потребностей и уровня подготовки.

Структура издания

Могу предположить, что вы начали читать эту книгу, потому что вас интересует использование GitHub Copilot в качестве потенциального ИИ-помощника в программировании. Или же вы хотите больше узнать о том, как он и другие похожие помощники позволяют работать быстрее и продуктивнее.

Чтобы вам было проще ориентироваться, расскажу о структуре книги.

В главах 1–3 излагаются основы, необходимые для понимания того, что такое Copilot и как взаимодействовать с ним в двух основных режимах: завершения кода и чат-интерфейса.

В главе 4 показаны различные способы взаимодействия с Copilot, позволяющие выполнять более автономное редактирование, а также описываются варианты использования режима агента, в котором Copilot переходит от *промпта*[1] (prompt) к конечному результату автоматически. Кроме того, в главе рассматривается уникальный интерфейс Copilot Vision и отладка с помощью Copilot.

В главах 5 и 6 показано, как Copilot помогает создавать развернутые наборы тестов и разные виды документации. Используя описанные в этих главах приемы, вы сможете упростить выполнение этих операций и получить больше времени на решение других задач программирования.

Глава 7 посвящена повышению актуальности и релевантности ответов Copilot.

В главе 8 вы научитесь использовать Copilot для выполнения некоторых менее распространенных, но полезных задач.

В главе 9 рассматривается чат-интерфейс Copilot в GitHub, а также то, как Copilot помогает получить больше информации о ваших проектах и упростить работу с GitHub Issues и pull-запросами.

Наконец, в главе 10 рассказывается о том, как расширить функциональность Copilot с помощью интеграции с GitHub Extensions и как создать собственное расширение.

Чтобы начать читать эту книгу, вам не обязательно что-либо знать о Copilot. Но я предполагаю, что у вас уже есть базовые навыки программирования и работы с Git и GitHub. В следующем разделе я более подробно написал, каким читателям может быть полезна эта книга.

---

[1] Промпт — это четко сформулированный запрос или инструкция, который пользователь передает искусственному интеллекту или нейросети для получения конкретного результата. — *Здесь и далее примеч. науч. ред.*

Для кого написана книга

Книга предназначена для любого читателя, который хочет больше узнать об использовании ИИ-помощников для разработки ПО. Да, книга посвящена GitHub Copilot, но многие примеры и процессы имеют аналоги в других ИИ-инструментах и средах разработки. Вы получите максимум пользы от книги, если у вас уже есть опыт программирования в IDE и базовые знания о GitHub. Буду откровенен: эта книга не научит вас программировать, работать в IDE или использовать GitHub. Но она поможет вам многократно повысить вашу продуктивность во всех областях за счет использования Copilot. Кем бы вы ни были: разработчиком, инженером по качеству, инженером по надежности или просто человеком, который хочет понять, как ИИ-помощник (такой как Copilot) может упростить его работу, я уверен, что вы найдете здесь полезную информацию и интересные идеи.

Вот категории читателей, на которые я ориентировался при работе над книгой и которым, по моему мнению, она пригодится:

- читатели, не имеющие опыта использования ИИ-помощников по программированию и желающие понять, что это вообще такое и как работать с такими инструментами;
- читатели, которые уже понимают основные концепции и процессы использования подобных ИИ-помощников и хотят понять, что им может предложить GitHub Copilot;
- читатели, желающие понять и оценить GitHub Copilot как инструмент, который может использоваться более широко в их организации или отрасли;
- читатели с опытом работы с GitHub Copilot, которые хотят использовать его возможности и функциональность по максимуму;
- читатели, которые хотят понять, как компенсировать недостатки GitHub Copilot;
- читатели, желающие разобраться в новых возможностях Copilot — например, в режиме агента;
- читатели, которые хотят создавать собственные расширения GitHub Copilot;
- читатели, которые работают в GitHub и хотят пользоваться средствами интеграции с Copilot.

К какой бы категории вы ни относились, я надеюсь, что книга будет вам полезна. Если вы прочитали ее и хотите высказать свое мнение о ней, то я буду рад получить обратную связь в виде отзывов. Описание того, как оседлать волну генеративного ИИ в программировании, начинается с главы 1. Приятного чтения!

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины и важные понятия.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, пути, имена файлов и их расширений.

`Моноширинный полужирный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Курсивный моноширинный шрифт`

Показывает текст, который должен быть заменен значениями, предоставляемыми пользователем или определяемыми контекстом.

Шрифт без засечек

Используется для обозначения элементов интерфейса, названий клавиш, папок, каталогов, URL и адресов электронной почты.

Так оформляется совет или предложение.

Так оформляется примечание.

Так оформляется предупреждение.

Использование исходного кода примеров

Дополнительные материалы (примеры кода, упражнения и т. д.) можно скачать по адресу https://github.com/techupskills/learning-github-copilot.

Если у вас возникнут вопросы технического характера по использованию примеров кода, то направляйте их по электронной почте на адрес support@oreilly.com.

Эта книга призвана помочь вам выполнять вашу работу. В общем случае все примеры кода вы можете использовать в своих программах и документации. Вам не нужно обращаться в издательство O'Reilly за разрешением, если вы не собираетесь воспроизводить существенные части кода, или если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, или если вы будете цитировать эту книгу либо примеры из нее, отвечая на вопросы заинтересованных лиц. Вам потребуется разрешение издательства O'Reilly, если вы хотите продавать или распространять примеры из книги либо добавить существенные объемы представленного в ней программного кода в документацию вашего продукта.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под такой ссылкой мы подразумеваем указание авторов, издательства и ISBN.

Чтобы получить разрешение на использование значительных объемов программного кода из книги, обращайтесь по адресу permissions@oreilly.com.

Благодарности

«GitHub Copilot. Программирование в паре с ИИ» — моя четвертая книга наряду со множеством технических руководств, отчетов, статей и других документов, написанных мной за последние годы. Превращение замысла в готовый продукт требует невероятных затрат времени и усилий не только от автора, но и от команды, участники которой воплощают проект в жизнь. Я безмерно благодарен людям, которые уделили много времени и сил, чтобы эта книга была опубликована и вы могли прочитать ее и, надеюсь, найти в ней много полезного.

В первую очередь хочу поблагодарить Джона Девинса (John Devins) — редактора-консультанта издательства O'Reilly, который поверил в эту книгу и ратовал за нее, как он делал для меня во многих других проектах. Джон, надеюсь, ты радуешься жизни на заслуженном отдыхе! Спасибо и Луизе Корриган (Louise Corrigan), продолжившей работу Джона. За время сотрудничества с O'Reilly я проникся настоящим уважением к труду редакторов-консультантов и научился ценить их вклад в поддержку значимых проектов, создание качественных учебных программ и поиск новых способов устранения пробелов в знаниях. Эти люди часто отходят на второй план, но их вклад играет ключевую роль в обучении, которое я и другие создатели контента проводят для участников программы O'Reilly Learning.

В самом начале работы над этой книгой я специально попросил Мишель Кронин (Michele Cronin) снова поработать со мной в качестве редактора проекта. Мишель была редактором еще одной моей книги — *Learning GitHub Actions*, и я знал, что она сможет довести этот проект до совершенства, придать ему законченный вид и обеспечить его успех. Мишель провела много часов, помогая устранять любые потенциальные препятствия, организуя работу рецензентов, собирая обратную связь и всегда оставаясь готовой ответить на любые вопросы. Кроме того, она делилась экспертными советами о том, как справиться с любыми возникающими затруднениями. Мишель сохраняла оптимизм и поддерживала меня на протяжении всего процесса выпуска книги. Я очень ценю все, что она сделала, и чрезвычайно благодарен ей за полученный опыт и полезные советы по подготовке книги.

Я должен упомянуть и многих других сотрудников O'Reilly. Шерон Уилки (Sharon Wilkey) и Джонатан Оуэн (Jonathon Owen) проделали исключительную работу, сделав текст более понятным и удобочитаемым. Кейт Дулли (Kate Dullea) помогала подготовить иллюстрации и проследить за тем, чтобы они были полезными и понятными. Спасибо Сьюзен Томпсон (Susan Thompson) за классную иллюстрацию на обложке и Дэвиду Футато (David Futato) за подготовку макета книги.

Многие выдающиеся эксперты из GitHub и других областей внесли свой вклад в виде времени и ресурсов, выступив в качестве научных редакторов книги. Несомненно, благодаря их усилиям книга существенно улучшилась. Большое спасибо

Эндрю Стеллману (Andrew Stellman), Стеффену Бьеркеносу (Steffen Bjerkenås), Чхае Метани (Chhaya Methani), Баладжи Дхамодхарану (Balaji Dhamodharan), Адитье Гоэлу (Aditya Goel), Сундипу Катте (Sundeep Katta), Джессу Мейлзу (Jess Males) и Тому Толли (Tom Taulli). Мне очень повезло, что эти замечательные люди пожелали выделить время в своих плотных рабочих графиках и поделиться обратной связью, идеями и предложениями, которые помогли книге стать лучше. Их совокупный опыт работы в технической сфере и внимание к деталям принесли огромную пользу. Отдельное спасибо Эндрю Стеллману за предисловие к книге — надеюсь, вы уделите несколько минут его прочтению.

Было бы несправедливо не поблагодарить Джея Циммермана (Jay Zimmerman), директора конференций NFJS (No Fluff Just Stuff), и Тодда Льюиса (Todd Lewis), председателя организации ATO (All Things Open), за возможность присутствовать и выступать на этих конференциях в прошедшие годы. Многочисленные презентации, которые я проводил на виртуальных семинарах и конференциях для NFJS и ATO, повысили качество этой книги. Уважаемые читатели, если у вас когда-нибудь появится возможность посетить одну из этих конференций или присутствовать на встрече или виртуальном мероприятии, спонсируемом этими организациями, то я рекомендую вам не упускать ее.

Спасибо моему коллеге и другу Бобу Пачеко (Bob Pacheco) за регулярное участие в технических и неформальных дискуссиях, которые помогали взглянуть на вещи под новым углом и разрядить обстановку.

И самое важное: хочу поблагодарить свою жену Анну-Марию (Anne-Marie), неизменно поддерживающую меня родственную душу, друга навсегда и спутницу жизни. Она и наши дети (Уокер, Чейз, Таннер и Кэти (Walker, Chase, Tanner и Katie)) постоянно напоминают мне, что наши близкие — это то, ради чего стоит жить и решать важные задачи, несмотря ни на что вокруг.

Наконец, я хочу поблагодарить вас, читателей, за то, что приобрели эту книгу и читаете ее. Искренне надеюсь, что она будет вам полезной и поможет добиться целей, которые вы поставили перед собой, каким бы ни был ваш уровень знакомства с GitHub Copilot.

От издательства

Мы выражаем огромную благодарность клубу рецензентов ИТ-литературы ReadIT Club за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы. Хотелось бы отметить, что мир нейронных сетей постоянно меняется, обучается и развивается, поэтому результаты, полученные вами, могут отличаться от тех, что вы увидите в книге.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство SprintBook, компьютерная редакция).

Мы будем рады узнать ваше мнение!

О научном редакторе русскоязычного издания

Татьяна Квист — ИИ-тренер в компании «Яндекс» с более чем пятилетним опытом работы в академической сфере. Автор пяти научных статей, в том числе международных, опубликованных в ведущих научных базах РИНЦ и Web of Science. Татьяна ежедневно работает с искусственным интеллектом. Она применяет инструменты на базе ИИ для оптимизации рабочих процессов и в повседневной жизни.

Ранее Татьяна редактировала научные статьи и преподавала информатику. В этой книге она помогает читателям разобраться в современной терминологии, делая сложные темы доступными и понятными.

ГЛАВА 1

Основы

Итак, мы начинаем! Я очень рад, что вы читаете эту книгу, и надеюсь, что она будет вам полезна. Такие ИИ-инструменты, как ChatGPT и ИИ-агенты, уже изменили и продолжают менять процесс нашего взаимодействия с приложениями. В свою очередь, GitHub Copilot и подобные ему инструменты уже изменили и продолжают менять процесс создания приложений программистами. GitHub Copilot умеет извлекать контекст из существующего кода или промптов на естественном языке, поэтому предоставляет более богатые и более мощные возможности генерации кода, чем другие приложения, созданные до него.

С этой книгой я помогу вам понять, как использовать функциональность GitHub Copilot для решения таких задач, как генерирование и автозавершение кода, автоматическая генерация тестов, преобразование и объяснение кода, работа с репозиториями, pull-запросами и задачами (issues), возникающими непосредственно в GitHub. Мы рассмотрим примеры использования Copilot в разных предметных областях и языках программирования. Вы научитесь эффективно применять его и писать промпты, позволяющие получать наилучший результат. Вы узнаете, как добавить в Copilot пользовательскую специализированную функциональность. А еще поймете, как он делает то, что делает, и почему вы иногда не получаете желаемый результат и как быть в таких ситуациях.

Copilot как общее название

В самом начале обсуждения необходимо сказать, что Copilot — популярное название для ИИ-приложений, которые собирают информацию, формулируют промпты и возвращают ответы и рекомендации. Например, у Microsoft есть система Office 365 Copilot (https://oreil.ly/DaSLT), обеспечивающая ту же функциональность для приложений Microsoft Office. Она анализирует контекст из Word, Teams, Outlook и так далее и предоставляет сводки, варианты ответов и предлагает другие полезные функции для взаимодействия.

В этой книге вместо полного названия GitHub Copilot я буду использовать просто Copilot, за исключением мест, где будут упоминаться другие системы, в названии которых тоже есть это слово. В таких случаях я буду указывать их полные официальные названия — например, Office 365 Copilot.

Однако прежде, чем начинать погружаться в тему, стоит ознакомиться с базовой информацией. Именно она и представлена в текущей главе. Мы начнем с общего описания GitHub Copilot. Затем рассмотрим базовую технологию, лежащую в его основе, общий принцип работы GitHub Copilot, особенности работы с ним и то, чем он отличается от таких инструментов, как ChatGPT. Кроме того, мы поговорим о том, где его взять и как установить.

Что такое GitHub Copilot

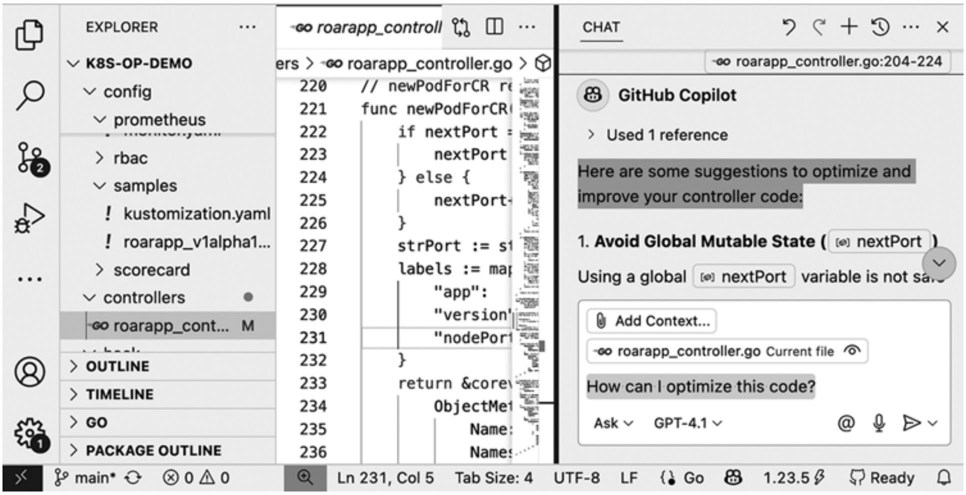
GitHub Copilot — облачная генеративная модель ИИ. Что же означают эти слова? В общем смысле ИИ можно определить как *компьютеры, решающие задачи, которые, как считалось ранее, могут решать только люди, поскольку имеют необходимые навыки и умеют рассуждать*. Кроме того, в последнее время у машин появилась возможность взаимодействовать с людьми так, как это делали бы сами люди, — благодаря использованию таких средств, как обработка естественного языка (Natural Language Processing, NLP), чат-интерфейсы и автоматизированная обработка и принятие решений с помощью ИИ-агентов.

СТОИТ ЛИ ОПАСАТЬСЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Когда речь заходит об ИИ, люди невольно мыслят образами поп-культуры: машины становятся разумными, обретают сознание и захватывают мир. Многие из нас, технических специалистов, понимают, что это нереально. Тем не менее программисты, пользующиеся такими инструментами, как GitHub Copilot, могут вообразить, что ИИ (в области программирования) может лишить их работы.

Безусловно, использование искусственного интеллекта уже привело к перестановкам на рабочих местах. И все же Copilot создавался не для этого. При всей кажущейся разумности, которую Copilot может демонстрировать в некоторых случаях, он может генерировать некорректный или неработоспособный код. И он по-прежнему нуждается в контроле со стороны человека. Я надеюсь, что благодаря этой книге Copilot займет почетное место в вашем инструментарии и поможет вам в работе над проектами и при выполнении ваших профессиональных обязанностей.

Слово *«облачный»* отсылает к способу, которым Copilot возвращает рекомендации и генерирует ответы. Речь идет об облачной среде под управлением GitHub, которая упрощает взаимодействие с ИИ-моделями. Определение *«генеративный»* отображает способность ИИ *генерировать* новые результаты на основе полученного контекста. Copilot может выдавать ответы и предложения для разработки, опираясь на контекст и подсказки из среды пользователя. Насколько хорошо он это делает? Зависит от нескольких факторов, о которых мы поговорим далее. В качестве короткого примера на рис. 1.1 показано, как Copilot предлагает оптимизации для проекта в Visual Studio (VS) Code.



```
// newPodForCR r
func newPodForCR
    if nextPort =
        nextPort
    } else {
        nextPort
    }
    strPort := s
    labels := ma
        "app":
        "version"
        "nodePor
    }
    return &core
        ObjectMe
            Name:
            Names
```

Here are some suggestions to optimize and improve your controller code:

1. **Avoid Global Mutable State (nextPort)**

Using a global nextPort variable is not sa

How can I optimize this code?

Рис. 1.1. Использование Copilot с проектом в VS Code

IDE по умолчанию

Описать все интегрированные среды разработки (IDE) в одной книге невозможно, поэтому в примерах, где фигурирует среда, будут использоваться VS Code или GitHub Codespace. Если вы работаете в другой IDE, то найдите в документации Copilot описание различий в использовании, элементах управления и т. д.

Как работает Copilot

Чтобы понять, как работает Copilot, необходимо иметь представление о некоторых компонентах, лежащих в основе функциональности его самого и других похожих ИИ-инструментов. В частности, эти компоненты определяют источник информации для формирования ответов Copilot и общий принцип его работы.

Copilot и многие ИИ-приложения получают свои данные от больших языковых моделей (large language models, LLM).

Большие языковые модели

LLM (без упоминания которых в наши дни не обходится ни одно обсуждение ИИ) — модели на базе искусственного интеллекта, обученные на огромных объемах существующих данных предсказывать следующие слова или другие виды контента (например, *токенов*), соответствующие заданным входным данным *(промпту)*.

От традиционных вычислительных моделей LLM отличаются способностью обрабатывать отформатированные данные или выдавать ответы на математические задачи, поскольку LLM учатся понимать контекст, синтаксис и структуру. Для этого они используют алгоритмы, которые анализируют огромное количество параметров, чтобы определить, какие слова или токены[1] лучше всего поставить следующими, исходя из их смысла. Таким образом, модели статистически формируют ответы, основываясь не только на входных данных, но и на форме их представления.

Прогностические способности модели изучаются и настраиваются на основании обширных наборов существующего контента. В процессе обучения модели ищут взаимосвязи между разными видами информации в любой заданной предметной области. С технической точки зрения, получив последовательность токенов в виде запроса, LLM способны анализировать синтаксис и структуру ввода, вычислять контекст на основе обучающих данных и прогнозировать последовательность токенов, которая с наибольшей вероятностью должна идти далее. Проще говоря, LLM, исходя из всех «переваренных» ими данных, умеют определять, что будет звучать естественно, и продолжить диалог.

НЕ ТОЛЬКО LLM

Помимо *традиционных* LLM, существуют и другие ИИ-модели — например, малые языковые модели (small language models, SLM) и модели логического вывода. Copilot постоянно обновляется, чтобы он мог использовать различные модели с разными возможностями и видами обучения, но для простоты я буду употреблять выражение «большие языковые модели».

Чтобы лучше понять, чем обучение LLM отличается от традиционной обработки, представьте, что изучаете новый язык, а потом общаетесь с человеком, для которого этот язык является родным. Вы можете заранее полистать словарь, запомнить отдельные слова и фразы. Но в общении с носителем языка необходимо также

---

[1] «Токен» — технический термин для элементарной единицы текста в модели (часто слово, подслово или символ).

понимать контекст того, что он говорит. Это поможет выбрать правильные слова или фразы для ответов. Кроме того, необходимо уметь сформулировать ответ в контексте, который будет осмысленным для собеседника. Язык имеет синтаксис и структуру, но они обретают смысл именно благодаря контексту общения. LLM могут определять контекст по входным данным и передавать соответствующий контекст в своих ответах.

Вернемся к Copilot. По умолчанию в нем используются LLM, разработанные и управляемые OpenAI (https://openai.com) — компанией, создавшей ChatGPT (https://chat.openai.com). В сотрудничестве с OpenAI компания GitHub разрабатывала Copilot в течение нескольких лет. Недавно были добавлены возможности использования других семейств моделей, включая модели Claude от Anthropic и Google Gemini. Во всех моделях общей характеристикой Copilot является ориентация на создание программного обеспечения, а языком общения — программный код.

Код и генеративный ИИ

Языки программирования по определению имеют конкретные обязательные синтаксические и семантические правила, в каждом языке разные. Когда вы пишете код на Python (а не на Go, к примеру), для создания программы используете разные токены и структуры. Тем не менее они подчиняются определенным правилам. Эти токены и структуры четко определены и образуют ограниченный набор. Copilot же предназначен для того, чтобы предоставлять рекомендации по программированию и сопутствующую информацию, которая соответствует синтаксису и структуре. Однако основное преимущество Copilot заключается в том, что он выдает ответы, имеющие отношение к тому, над чем работает программист или о чем спрашивает в промпте. Контекст, который передается в Copilot, может поступать из нескольких источников, включая следующие:

- код, написанный в редакторе в среде разработки;
- взаимодействие с моделью с помощью промптов на естественном языке (также известных как *чат-модели*);
- типичные операции разработки непосредственно в GitHub (например, pull-запросы).

Все эти варианты взаимодействия будут рассмотрены в книге. Независимо от интерфейса контекст (код, инструкции (directives), вводимые данные или элементы GitHub, с которыми вы работаете) преобразуются в *промпт* — вашу реплику в диалоге, на которую должен отреагировать ИИ. Обработка входных данных и формирование ответа на основании контекста, обучения и возможностей модели — *генеративная* часть работы Copilot как *генеративного ИИ*. Ответом могут быть предлагаемый код, ответ на вопрос или пошаговые инструкции. Это то, что, по мнению ИИ, завершает код, соответствует промпту или отвечает на запрос.

ЧАТ-МОДЕЛИ

Как мы чаще всего представляем себе программирование? Программист сидит перед терминалом и вводит код в IDE. Но прежде чем вводить код, он часто должен получить ответы на какие-то вопросы или что-то понять. Например, при изучении нового языка программист может сомневаться в том, как реализовать некую структуру данных или управляющую конструкцию. А может, он захочет узнать, как открывать файлы или выполнять операции чтения/записи в базе данных. А затем ему нужно будет разобраться в различиях версий сторонней платформы или выяснить, не устарела ли та или иная функциональность. Обычно такие сведения вы узнавали у коллег или искали на Stack Overflow. Но если вместо этого вы пользуетесь ИИ, то должны уметь задать ему вопросы или давать указания (то есть иметь возможность *вести диалог*).

Типичные *общедоступные* реализации генеративного ИИ (такие как ChatGPT) стали очень популярными потому, что могут общаться и отвечать на вопросы понятным для человека образом. В Copilot это возможно благодаря *Copilot Chat*. В чат-интерфейсе можно дать Copilot указание выполнить некоторые операции (например, «Сгенерируй юнит-тесты для кода») или задать интересующие вас вопросы (допустим, «Какая версия Kubernetes является текущей?» и даже «Что делает этот код?»). Формирование ответов на основе инструкций или вопросов в чат-интерфейсе — еще один пример *генеративной* составляющей *генеративного* ИИ.

В качестве аналогии представьте, что описываете в Zoom симптомы доктору, чтобы он попытался поставить диагноз, опираясь на свой многолетний опыт. Или что по телефону описываете автомеханику проблему с вашей машиной, чтобы он мог предложить решение. В таких случаях успех сильно зависит от ваших навыков коммуникации и контекста, который вы предоставляете, а также от квалификации профессионала, к которому вы обратились. В результате общения вы можете получить советы по решению проблемы (и некоторые из них могут оказаться неактуальными) или узнать, что профессионалу не хватает контекста или понимания, чтобы помочь вам. Генеративный ИИ работает по тому же принципу.

Когда Copilot дает вам рекомендации, пока вы пишете код, элементы среды разработки формируют контекст, в котором будет создаваться промпт для модели ИИ. Copilot может генерировать предложения, которые соответствуют стилю программирования, используемому в файлах вашего проекта. Это может быть как хорошо, так и плохо.

Предложения Copilot часто соответствуют стилю программирования реальных пользователей, и это может быть хорошо, если эти стили соответствуют лучшим практикам, и плохо, если Copilot слепо копирует неудачные практики из кодовой базы. В последнем случае вы можете привыкнуть использовать ограниченные и неэффективные способы программирования либо же вам придется изучать более обширный набор рекомендаций, чтобы найти подходящий вариант. На ответы Copilot влияют количество и качество примеров, которые он сможет извлечь из вашей среды, а также степень его обученности.

Таким образом, генеративный ИИ, используемый в Copilot, может оказаться очень полезным. Но следует помнить, что на его рекомендации по написанию кода будет влиять контекст, с которым приходится работать Copilot, и данные, которые он использует для обучения. Вы можете обратить этот факт себе на пользу, предоставляя Copilot больше примеров стиля предложений, которые он должен выдавать. Но такая зависимость может стать недостатком, если контекст, доступный Copilot, ограничен. В дальнейших главах мы обсудим, как помочь Copilot выдавать наилучшие результаты.

Далее мы кратко рассмотрим общую схему работы Copilot, чтобы вы могли лучше понять, как он взаимодействует с контекстом при формировании ответа.

Общая схема работы Copilot

Чтобы понять принцип работы Copilot, можно рассмотреть основной процесс с точки зрения работы в одной из поддерживаемых IDE. В настоящее время в список поддерживаемых сред разработки для Copilot входят Visual Studio, VS Code, NeoVim, все представители семейства JetBrains и другие среды. Помимо этого, Copilot работает и в некоторых других средах, таких как в GitHub Codespaces (https://oreil.ly/QyEks).

Кроме того, GitHub выпустил пакет Copilot Language Server SDK (https://oreil.ly/6rDHq), с помощью которого GitHub Copilot можно интегрировать в любой редактор или IDE. А значит, в будущем можно ожидать появления новых приложений, интегрированных с Copilot.

Codespaces

GitHub Codespaces использует виртуальные машины (virtual machines, VM), работающие на облачной платформе, которые предоставляют полнофункциональную и настраиваемую среду разработки для пользователей GitHub. Интерфейс Codespaces напоминает интерфейс VS Code; эта среда может дополняться теми же расширениями, которые используются в VS Code, через браузер или путем подключения к разным IDE. GitHub предоставляет CodeSpaces как дополнительный сервис.

Когда вы пишете код и запрашиваете у Copilot варианты завершения, он автоматически сканирует различную информацию, чтобы получить контекст того, над чем вы работаете.

Текущий файл

Текущий файл, который редактируется пользователем, — ключевой источник контекста для Copilot.

Имя текущего активного файла

По описательному имени (например, `TestConfig.go`) Copilot может понять, что должен делать код.

Контент до и после текущей позиции курсора

Copilot может учитывать контекст из кода и комментариев, расположенных непосредственно до и после текущей позиции курсора в файле. Эта информация поможет Copilot решить, что именно нужно вставить, и/или понять, какие части кода отсутствуют.

Комментарии

Как и программисты-напарники или рецензенты кода, Copilot может по комментариям понять, что делает уже существующий код и что должен делать еще не написанный. Добавление комментариев — один из основных способов предоставления контекста для Copilot: чем точнее и подробнее комментарии, тем выше вероятность того, что предложенный им код будет актуальным.

Другие открытые в редакторе файлы

Copilot использует код, содержащийся в любых открытых файлах, как контекст. Это ключевой момент для сбора информации о текущей задаче и дополнения информации в модели. Так, одна из стратегий работы с устаревшей функциональностью LLM — открытие в редакторе файла с примером замены неактуальной функции. По этому примеру Copilot сможет понять, какие альтернативы будут предпочтительными, вместо того чтобы полагаться на устаревшие подходы, использовавшиеся при обучении LLM.

Локальный индекс

Copilot автоматически анализирует большинство файлов в проекте, открытом в таких IDE, как VS Code, и создает расширенный локальный индекс для проекта.

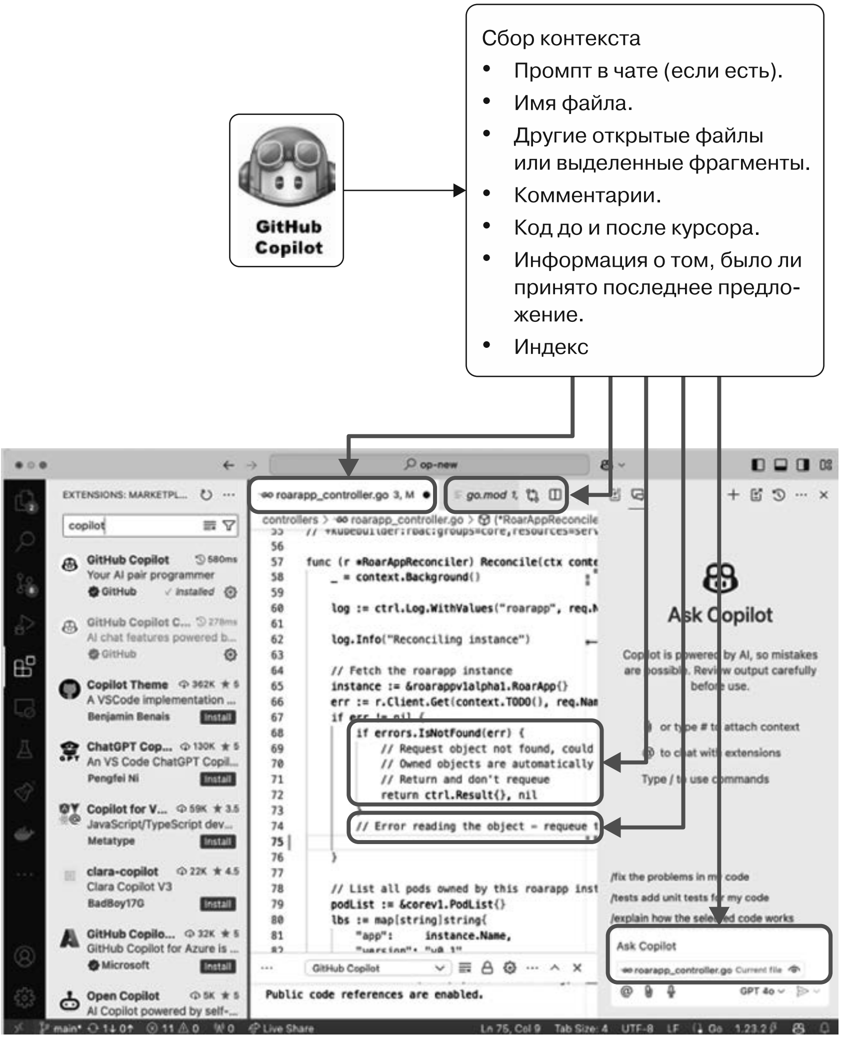
ИНДЕКСИРОВАНИЕ

В контексте Copilot *индексированием* называется процесс сканирования и организации кодовой базы для создания структуры данных, обеспечивающей быстрый и точный поиск. Наличие индекса позволяет Copilot предоставлять более точные и контекстно зависимые предложения и ответы по коду.

Локальный индекс, создаваемый и обновляемый автоматически при внесении изменений, хранится на машине пользователя. При использовании в GitHub (см. главу 9) Copilot создает удаленный индекс, связанный с репозиторием.

Если вы используете чат-интерфейс, то обычно он заранее заполняется контекстом из файла, выделенного фрагмента или команды терминала — в зависимости от того, над чем вы работали в последнее время. Впрочем, этот контекст можно изменить перед отправкой промпта (подробнее об этом поговорим в главе 3).

Таким образом, когда вы используете один из интерфейсов с установленным и активизированным Copilot, он собирает информацию из ряда источников по мере того, как вы вводите код (рис. 1.2).



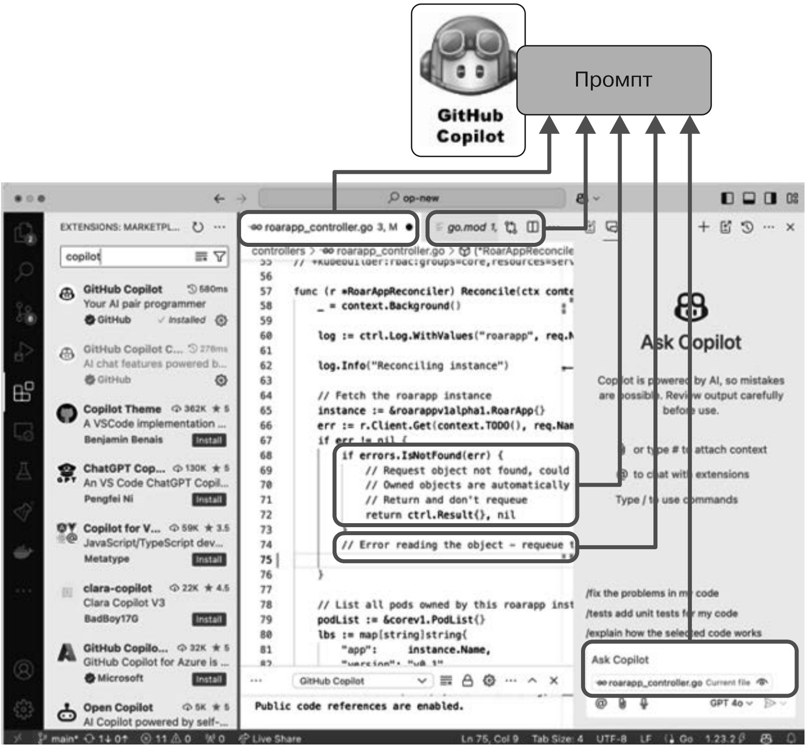
Сбор контекста
- Промпт в чате (если есть).
- Имя файла.
- Другие открытые файлы или выделенные фрагменты.
- Комментарии.
- Код до и после курсора.
- Информация о том, было ли принято последнее предложение.
- Индекс

```
// +kubebuilder:rbac:groups=core,resources=serv

func (r *RoarAppReconciler) Reconcile(ctx conte
    _ = context.Background()

    log := ctrl.Log.WithValues("roarapp", req.N

    log.Info("Reconciling instance")

    // Fetch the roarapp instance
    instance := &roarappv1alpha1.RoarApp{}
    err := r.Client.Get(context.TODO(), req.Nam
    if err != nil {
        if errors.IsNotFound(err) {
            // Request object not found, could
            // Owned objects are automatically
            // Return and don't requeue
            return ctrl.Result{}, nil
        }
        // Error reading the object - requeue t

    }

    // List all pods owned by this roarapp inst
    podList := &corev1.PodList{}
    lbs := map[string]string{
        "app":     instance.Name,
        "version": "v0.1"
```

```
Public code references are enabled.
```

Рис. 1.2. Сбор контекста из среды IDE

Контекст обрабатывается и в конечном счете отправляется на GitHub, где синтезируется промпт (рис. 1.3).



```
func (r *RoarAppReconciler) Reconcile(ctx conte
    _ = context.Background()

    log := ctrl.Log.WithValues("roarapp", req.N

    log.Info("Reconciling instance")

    // Fetch the roarapp instance
    instance := &roarappv1alpha1.RoarApp{}
    err := r.Client.Get(context.TODO(), req.Nam
    if err != nil {
        if errors.IsNotFound(err) {
            // Request object not found, could
            // Owned objects are automatically
            // Return and don't requeue
            return ctrl.Result{}, nil

        // Error reading the object - requeue t

    }

    // List all pods owned by this roarapp inst
    podList := &corev1.PodList{}
    lbs := map[string]string{
        "app":     instance.Name,
```

```
Public code references are enabled.
```

Copilot is powered by AI, so mistakes are possible. Review output carefully before use.

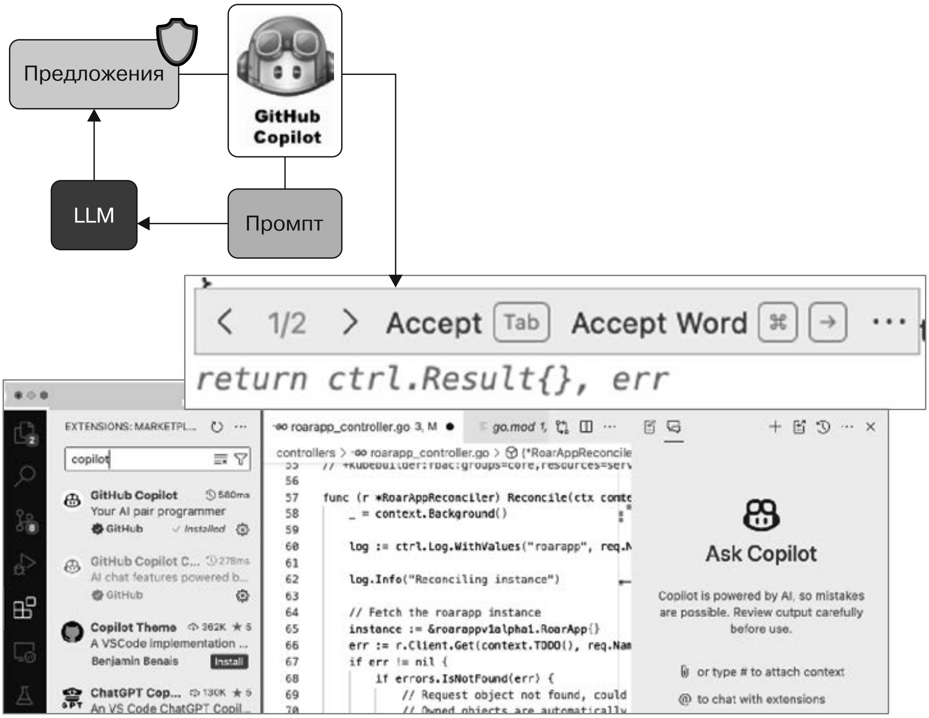
/fix the problems in my code
/tests add unit tests for my code
/explain how the selected code works

Рис. 1.3. Синтезирование промпта

Затем промпт передается через GitHub большой языковой модели, которая возвращает возможные завершения или ответы. После того как по промпту будут сгенерированы результаты, системы Copilot на GitHub выполняют дополнительную обработку результата (подробнее об этом поговорим позднее). Далее ответ возвращается в IDE, где вы можете оценить его и решить, как действовать дальше. Схема завершения кода показана на рис. 1.4.

Процесс продолжается и повторяется при дальнейшем взаимодействии пользователя с Copilot. Таким образом, Copilot действует как ассистент, помогающий вам разобраться с задачей, над которой вы работаете, будь то написание шаблонного кода, поиск сложного алгоритма, генерация данных или запросов, написание модульных тестов или изучение возможностей нового языка программирования.

Более подробно о том, как это происходит, мы поговорим в последующих главах, когда будем обсуждать работу с Copilot. А пока рассмотрим нюансы его использования, которые помогут вам в разработке программного обеспечения.



```
return ctrl.Result{}, err
```

```

func (r *RoarAppReconciler) Reconcile(ctx conte
    _ = context.Background()

    log := ctrl.Log.WithValues("roarapp", req.N

    log.Info("Reconciling instance")

    // Fetch the roarapp instance
    instance := &roarappv1alpha1.RoarApp{}
    err := r.Client.Get(context.TODO(), req.Nam
    if err != nil {
        if errors.IsNotFound(err) {
            // Request object not found, could
```

Copilot is powered by AI, so mistakes are possible. Review output carefully before use.

Рис. 1.4. Схема процесса от получения промпта до выдачи ответов

Особенности использования Copilot

Итак, теперь, когда вы знаете, что такое Copilot и как он работает, необходимо озвучить ключевой момент: Copilot, как и любой ИИ, может давать неверные или неполные ответы. Он может ошибаться или выдавать неожиданные результаты. Такое бывает нечасто, но при работе с ним нужно помнить о некоторых важных критериях. Это:

- актуальность;
- релевантность;
- полнота;
- точность;
- конфиденциальность;
- безопасность.

О них мы и поговорим в текущем разделе. Кстати говоря, эти критерии играют важную роль при взаимодействии не только с Copilot, но и с любым другим современным ИИ-инструментом, помогающим решать те или иные задачи.

Актуальность

Под *актуальностью* понимается свежесть рекомендаций Copilot. Может показаться странным, что этот критерий поставлен на первое место, но он может быть связан со всеми остальными.

Copilot полагается на модели, которые прошли обучение в некий момент времени, поэтому его данные основаны на информации, которая была актуальной в тогдашних данных для обучения. Например, если модель, используемая Copilot, прошла обучение два года назад, то может не знать об изменениях, произошедших с того момента.

При повседневном использовании Copilot может выдавать неактуальные предложения и ответы. Результаты могут содержать устаревший код, который не работает в вашем компиляторе или интерпретаторе. Вы можете получить предложение со ссылкой на версию зависимости с известной уязвимостью или ответ, содержащий устаревшее решение.

Можно спросить в чат-интерфейсе Copilot: «Устарело ли ...?» — и получить ответ, что нет, *хотя на самом деле это не так*. Copilot дает ответы, актуальные на момент обучения модели. Или можно спросить: «Какая версия ... является текущей?» — и получить результат двухгодичной давности. Наверняка вы понимаете, к каким проблемам это может привести. Кроме того, результаты могут значительно разниться в зависимости от того, какую модель (из числа поддерживаемых Copilot) вы выбрали для текущего взаимодействия.

ПОЛУЧЕНИЕ ИНФОРМАЦИИ О ТЕКУЩЕЙ ВЕРСИИ

Допустим, вы спрашиваете Copilot в чат-интерфейсе: «Какая версия ... является текущей?» В большинстве случаев ответ будет изменяться в зависимости от модели, которую вы используете. В одних случаях ответ будет содержать оговорку «на дату обучения» или что-то подобное. В других Copilot порекомендует обратиться к документации для получения актуальной информации.

Нельзя рассчитывать, что модель будет знать, какая версия является новейшей, поэтому задавать такие вопросы обычно бессмысленно. *Иногда* полученная информация может быть полезной (например, она даст представление о том, какая версия была актуальной на момент обучения модели), но правильность и детализация ответа будет зависеть от модели.

В главе 7 мы обсудим методы предоставления Copilot более актуального контекста для устаревших элементов вашей среды, что повышает вероятность выдачи более актуальных ответов.

Релевантность

Предложения Copilot и ответы в чате основываются на моделях типа LLM, обученных на больших объемах кода. Если вы используете модели OpenAI по умолчанию, то этой кодовой базой является общедоступный код, размещаемый на GitHub. Считается, что GitHub располагает самой полной коллекцией репозиториев программного обеспечения с открытым кодом. Они содержат код, написанный на самых популярных языках программирования нашего времени, таких как Python, Go и JavaScript.

Исходя из процесса обучения, логично предположить, что Copilot будет наиболее эффективен при работе с популярными языками программирования или фреймворками, представленными в кодовой базе, которая использовалась для обучения. Чем больше язык или фреймворк представлен в этой коллекции (репозитории GitHub для моделей OpenAI), тем больше источников для обучения получит Copilot. Для сравнения вспомните учебные предметы, изучению которых вы уделяли больше всего времени в школе, — именно их вы знаете лучше всего.

Если вы используете для работы менее популярный язык или фреймворк, то предложения Copilot могут быть не столь полезными. Это не означает, что вы не получите актуальных рекомендаций и ответов, но они могут быть менее полезными в плане получения точных ответов на ваши конкретные запросы.

Ключевой фактор, неподконтрольный Copilot, — объем контекста, с которым ему приходится работать. Например, если код содержит функцию `ParseData` или вы предоставляете слишком общий или неоднозначный промпт (например, «Создай функцию для разбора данных») без дополнительного контекста, то и результаты, возвращаемые Copilot, с большой вероятностью будут слишком общими.

Полнота

Генеративный ИИ иногда может возвращать неполные или неработоспособные предложения. В чат-интерфейсе это происходит редко, но функция завершения кода Copilot достаточно часто возвращает набор предложений, которые представляют собой частичное решение.

Проблема, о которой мы говорили в конце предыдущего подраздела, посвященного релевантности, актуальна и здесь. Если ваш код или запрос слишком общий или неоднозначный, то Copilot может не хватить контекста для формирования полного результата.

Бывают и другие ситуации: вы можете предоставить конкретный контекст, но Copilot предложит только комментарий или первую строку функции. Обычно так происходит при работе с предложениями непосредственно в IDE, а не в чат-интерфейсе. Возможно, вам придется предоставить Copilot расширенный промпт или передать больше контекста. Иногда удается *подтолкнуть* Copilot в нужном направлении, предоставив дополнительный промпт с уточнением (например, введя ключевое слово), и этого будет достаточно для формирования более полного предложения. В других случаях Copilot может выдать просто пустую строку, а когда вы примете этот ответ, продолжит генерировать код.

Copilot предлагает возможности для получения других предложений, если текущий вариант вас не устраивает, но полезность этих альтернатив может быть ограниченной. Более подробно эти подходы мы рассмотрим в главе 2.

Точность

Если не считать простейших случаев, вы не можете заранее считать, что любой ответ Copilot — заведомо правильный или лучший. *Его ответы всегда следует тщательно проверять*. Нетрудно найти истории об использовании ИИ в разных областях, когда результаты содержали ссылки на несуществующие ресурсы. В качестве примера можно привести судебные отчеты, в которые включали ссылки на ненастоящие дела. Некоторые люди пытались использовать ИИ для планирования отпусков и получали отличные результаты — но с гостиницами, которых на самом деле не существовало.

Аналогично Copilot может вернуть ответ, который имеет правильный синтаксис, но ссылается на конструкции или переменные, не существующие в коде. Это одна из разновидностей *галлюцинаций ИИ*: искусственный интеллект в качестве решения или предложения выдает информацию, которая неверна, вымышлена или по иным причинам не основана на реальности. Подобные инциденты встречаются редко, но все же не исключаются. Иногда они происходят из-за того, что Copilot приходится создавать предложения для элементов, которые еще не существуют. Например, если вы спросите, как открыть и записать файл, не указав его имя, то Copilot может использовать имя, не имеющее смысла в коде вашего примера.

Независимо от обстоятельств пользователь всегда несет ответственность за проверку точности ответов Copilot. Это сродни проверке кода, добавленного вашим коллегой: необходимо убедиться, что этот код правильный и не причинит вреда.

Промпты и точность

Результаты Copilot часто можно улучшить, составив более точный промпт для ИИ. О создании промптов мы поговорим позже.

Конфиденциальность

Еще один критерий работы с Copilot — конфиденциальность данных. Как упоминалось ранее, Copilot считывает информацию из нескольких источников (включая открытые и текущие файлы) и передает ее на GitHub, чтобы были созданы промпты для LLM. Это означает, что часть данных передается по сети и обрабатывается вне вашего контроля. Может показаться, что это создает потенциальный риск безопасности. Но Copilot позволяет при регистрации указать, хотите ли вы разрешить GitHub добавлять вашу контекстную информацию в данные, чтобы улучшить работу Copilot (см. нижнюю часть рис. 1.5). Если вы откажетесь, то информация все равно будет собрана, однако GitHub будет использовать ее строго в рамках трех областей, касающихся взаимодействия с пользователем, промптов и предложений по коду.

Policies

Suggestions matching public code

GitHub Copilot can allow or block suggestions matching public code. See the GitHub Copilot documentation to learn more.

Blocked

☐ **Allow GitHub to use my code snippets for product improvements ***

Allow GitHub, its affiliates and third parties to use my code snippets to research and improve GitHub Copilot suggestions, related models and product features. More information in Privacy FAQ.

Рис. 1.5. Настройки конфиденциальности данных при регистрации

Copilot Trust Center

В GitHub для решения проблем, связанных с безопасностью, существует Trust Center (https://oreil.ly/0GM3-).

Copilot работает с пользовательскими данными в трех областях: взаимодействие с пользователем, промпты и предложения по коду.

Взаимодействие с пользователем

Это данные о работе пользователя с Copilot (телеметрия). Могут содержать информацию о том, принимает ли он предложения Copilot или отклоняет их, как использует чат-интерфейс, а также такие показатели, как задержки и сообщения об ошибках.

Промпты

Как упоминалось ранее, контекстная информация берется из вашей среды или запроса из чата, переданного GitHub.

Предложения по коду

Предлагаемые варианты завершения кода, возвращаемые Copilot, и/или ответы на запросы в чате.

Данные о взаимодействии с пользователем отслеживаются по умолчанию и могут содержать анонимизированные данные, хотя возможны нюансы в зависимости от тарифного плана Copilot (подробнее — в документации Trust Center (https://oreil.ly/rC8Rm)). Существуют различные механизмы для шифрования данных как при передаче, так и при хранении. В GitHub есть система контроля, позволяющая пользователям жестко ограничить доступ к данным на их стороне.

Репозитории GitHub и юридические аспекты

Законность использования публичных репозиториев GitHub для обучения некоторых LLM оспаривалась как в рамках общественного мнения, так и в судах. Мы не будем рассматривать эту тему или комментировать эти аспекты первоначального процесса обучения.

Существует еще одна проблема, которую часто поднимают в связи с Copilot при использовании модели, обученной на лицензированных репозиториях: Copilot может генерировать предложения, дублирующие код из этой кодовой базы. Это означает, что пользователи, добавив дублированный код в свою работу, могут непреднамеренно нарушить условия лицензирования и права на интеллектуальную собственность. В Copilot есть настройка, которая позволяет пользователям и администраторам блокировать использование совпадений с публичным кодом, если они генерируются в процессе работы ИИ (см. верхнюю часть рис. 1.5). Если задать эту настройку, то Copilot предупредит вас о совпадениях и отфильтрует их.

ЗАЧЕМ РАЗРЕШАТЬ ИСПОЛЬЗОВАНИЕ ЗАИМСТВОВАНИЙ

Под *заимствованиями* (matching content) понимаются точные или почти точные соответствия между кодом в репозиториях GitHub и кодом, предлагаемым Copilot. Возможно, у вас возник вопрос: зачем добавлять настройку, разрешающую использование заимствований, какая от этого польза? По факту ее может и не быть. Но когда вы просто изучаете, как выполняется то или иное действие, и не планируете использовать совпадающий код (или хотите использовать его на условиях лицензии, если он представляет собой лучшее решение), такой вариант может оказаться приемлемым.

Безопасность

И последний критерий, который мы должны обсудить, — безопасность результатов, возвращаемых Copilot. Это постоянная проблема для любого продукта или приложения, используемого в наши дни. Чтобы ее решить, нужно начать с соблюдения безопасных практик программирования.

Ранее я упоминал, что, после того как Copilot получает потенциальные результаты от LLM, GitHub выполняет их дополнительную обработку. В частности, он использует алгоритмы быстрого поиска возможных уязвимостей. GitHub не запускает полноценное сканирование безопасности результатов, поскольку это заняло бы слишком много времени. Вместо этого быстро отслеживает закономерности, характерные для уязвимостей и/или небезопасных практик программирования. Если они обнаруживаются, то предлагаемое решение помечается как требующее особого внимания.

Даже при наличии всех этих мер нет гарантий, что проблема не возникнет. Обработка, описанная в предыдущем абзаце, не эквивалентна полноценному сканированию с помощью специализированного инструмента, ориентированного на поиск уязвимостей, поэтому результаты, полученные от Copilot, все равно должны пройти все проверки безопасности, которые вы используете для любого другого кода.

Таким образом, при использовании генеративного ИИ неукоснительно соблюдайте четкое требование: всегда проверяйте и оценивайте любые его предложения по коду. Не рассчитывайте, что он полностью и правильно понял контекст. И во всех случаях, кроме самых простых, не считайте, что результат идеален.

То же самое можно сформулировать иначе. Copilot часто называют ИИ-напарником, но он не всегда так же хорошо понимает ваш код и знаком с ним, как реальный напарник-человек. Вместо этого лучше относиться к Copilot как к опытному программисту, который только что стал участником проекта. Такой программист может создавать полезный код, основываясь на своих наблюдениях и информации, которая ему доступна, но он не знает всего контекста или истории проекта. В результате необходимо тщательно проверять, что написанный этим человеком код точен, безопасен и пригоден для слияния. К ответам и предложениям Copilot следует относиться так же.

Как и при работе с новым участником команды, чем больше информации вы предоставите (кода, который должен использоваться для получения контекста, или конкретных указаний), тем лучше результат. Если вы дадите Copilot больше примеров кода для анализа или конкретизируете промпт в чате, то это поможет ИИ выдать улучшенный результат.

Когда вы начинаете работать с таким помощником, как Copilot, или только планируете это, может возникнуть другой вопрос: почему бы не использовать ChatGPT или аналогичный чат-бот для генерирования кода? Отвечу: да, это возможно. Но эти два подхода и интерфейса имеют принципиальные различия. В следующем разделе я привел краткое сравнение ChatGPT и Copilot.

Copilot и ChatGPT

Чем же Copilot или аналогичный помощник-программист отличается от ChatGPT или других подобных чат-ботов, ведь и те и другие для выдачи ответов используют одни и те же модели? Основное различие заключается в том, что Copilot создан только для работы в области программирования и его функциональность рассчитана именно на это. ChatGPT, напротив, может применяться практически в любых областях и не обеспечивает тот уровень интеграции со средами разработки, который предоставляет Copilot. В табл. 1.1 приведены различия по более конкретным категориям.

Таблица 1.1. Сравнение Copilot с ChatGPT

Категория	GitHub Copilot	ChatGPT
Основная функция	Понимание и генерирование кода или промптов, связанных с кодом	Понимание и генерирование текста на любом естественном языке
Основной пользовательский интерфейс	Редакторы кода, чат	Чат
Разработчик	GitHub в сотрудничестве с OpenAI	OpenAI
Основные варианты использования	Написание и доработка кода и документации	Ответы в разговорном стиле с генерированием текста
Модель ценообразования	Подписка на основании отслеживания использования для премиальных моделей	На основании использования и подписки
Обучающие данные	Репозитории кода, документация	Разнообразный текстовый контент
Открытые API	Ограничиваются телеметрией, мониторингом и управлением лицензиями	Широкая поверхность API для взаимодействия

В целом, Copilot можно считать очень узкоспециализированной реализацией генеративного ИИ — в противоположность более широкой (универсальной) реализации и функциональности ChatGPT и других аналогичных чат-инструментов. Конечно, ChatGPT можно использовать для генерирования кода, но ему не хватает интеграции со средами разработки, фокусировки на программировании, а также поддержки GitHub и функций, имеющихся в Copilot.

До настоящего момента мы рассматривали Copilot как единое приложение, но в действительности он насчитывает пять конфигураций. В завершение этой вводной главы я расскажу о тарифных планах Copilot.

Тарифные планы Copilot

На момент написания книги Copilot предлагает пять планов: Free, Pro, Pro+, Business и Enterprise. Чтобы оценить различия между ними, необходимо понимать некоторые часто встречающиеся термины.

Завершение кода

Код, предлагаемый ИИ для окончания кода, над которым вы работаете в IDE.

Чат-запрос

Промпт (указание или вопрос), передаваемый Copilot через чат-интерфейс.

Режим агента

Режим работы, в котором Copilot, получив промпт, самостоятельно планирует свои действия, выполняет их и последовательно вносит в файлы изменения, необходимые для выполнения задачи.

Запрос режима агента

Запрос, переданный в Copilot для выполнения задачи в режиме агента.

Доступные модели

Большие языковые модели, к которым Copilot имеет доступ, из всего набора используемых им LLM. Список моделей, поддерживаемых на данный момент, добавлен в документацию Copilot (https://oreil.ly/icFQK).

Запросы уровня премиум

Взаимодействия, в которых сложные ИИ-модели используются для таких операций, как Copilot Chat, режим агента, ревью кода или расширения. Они потребляют больше вычислительных ресурсов и учитываются отдельно от стандартных предложений по завершению кода.

ЗАПРОСЫ УРОВНЯ ПРЕМИУМ

Если вы пользуетесь расширенной моделью, то учтите, что некоторые функции Copilot требуют дополнительных вычислительных мощностей. Они относятся к категории *запросов уровня премиум*. В сущности, любая разновидность взаимодействия с моделью может быть таким запросом в зависимости от тарифного плана и модели.

Если вы на бесплатном плане, то каждый из ваших запросов считается премиальным. Если вы выбрали платный план, то запросы, использующие базовую модель по умолчанию (GPT-4o на момент написания книги), считаются обычными, в отличие от запросов, использующих любые другие модели.

Стоит заметить, что отношение между обычными запросами и относящимися к уровню премиум не всегда один к одному. Отдельный запрос, обращенный к некоторым моделям, может рассматриваться как менее чем один или более чем один запрос уровня премиум в зависимости от модели. Коэффициенты текущих моделей можно найти в документации Copilot (https://oreil.ly/ahzPJ).

Например, если вы пользуетесь планом Pro, то при выполнении 20 чат-взаимодействий с моделью o1 вы используете 200 (20 × 10) из 300 выделенных запросов уровня премиум на месяц (на момент написания книги). (Количество запросов уровня премиум для разных планов приводится в табл. 1.2.)

Если вы исчерпали лимит запросов уровня премиум на платном плане, но хотите продолжить работу, то можете приобрести дополнительные запросы этого уровня по цене $0,04 за один такой запрос.

В табл. 1.2 описаны доступные тарифные планы и их ключевые различия (с учетом приведенных выше терминов).

Таблица 1.2. Сравнение тарифных планов GitHub Copilot

Параметр	Free	Pro	Pro+	Business	Enterprise
Целевые пользователи	Отдельные разработчики, которые хотят исследовать возможности Copilot	Отдельные разработчики, которые хотят иметь неограниченный доступ к Copilot, но которым не нужно использовать все модели ИИ	Отдельные разработчики, которые хотят свободно пользоваться всеми функциями Copilot без ограничений и иметь доступ ко всем моделям ИИ	Организации и предприятия, которым нужен неограниченный доступ к базовой функциональности Copilot и функциям управления бизнес-процессами, но не нужен доступ ко всем моделям ИИ	Организации и предприятия, которым нужны: • неограниченный доступ к базовой функциональности Copilot и расширенным функциям управления бизнес-процессами; • возможность свободно пользоваться всеми функциями Copilot без ограничений; • доступ ко всем моделям ИИ; • приоритетная техническая поддержка и соблюдение корпоративных стандартов
Завершение кода, чат-запросы и запросы режима агента	Пятьдесят чат-запросов или запросов режима агента в месяц (считаются премиальными). Две тысячи запросов на завершение кода в месяц	Без ограничений	Без ограничений	Без ограничений	Без ограничений
Какие модели доступны	Небольшое подмножество моделей	Большее количество моделей	Все модели	Большее количество моделей	Все модели
Запросы уровня премиум, в месяц	50	300	1500	300	1000

Параметр	Free	Pro	Pro+	Business	Enterprise
Дополнительные возможности	Ограниченное ревью кода для выделений в VS Code	• Ревью кода. • Сводки pull-запросов	• Ревью кода. • Сводки pull-запросов	• Ревью кода. • Сводки pull-запросов. • Управление пользователями и показатели. • Конфиденциальность данных. • Правовая защита, которую компания-разработчик предоставляет пользователям своего продукта	• Ревью кода. • Сводки pull-запросов. • Управление пользователями и показатели. • Конфиденциальность данных. • Правовая защита, которую компания-разработчик предоставляет пользователям своего продукта
Администрирование	Индивидуальное	Индивидуальное	Индивидуальное	Корпоративное	Корпоративное
Стоимость	Бесплатно	$10/месяц или $100/год (с30-дневным бесплатным пробным периодом)	$39/месяц или $390/год	$19/месяц на пользователя	$39/месяц на пользователя

Актуальность информации

Информация в табл. 1.2 актуальна на момент написания книги. Обновленные данные можно получить на странице с описанием возможностей Copilot (https://oreil.ly/ptoVy) или на странице с описанием планов Github Copilot (https://docs.github.com/ru/copilot/get-started/plans).

Безусловно, стоимость — важный фактор. Если вы индивидуальный пользователь, который время от времени использует Copilot в своей IDE и не нуждается в дополнительной функциональности, то, возможно, вам подойдет план Free. Индивидуальному пользователю, который хочет регулярно работать с Copilot и использовать дополнительные возможности, могут подойти планы Pro или Pro+.

На уровне организации или сообщества, если вам нужна дополнительная возможность легко назначать лицензии и управлять ими для множества пользователей, а также расширенные средства администрирования, используйте план Business. А если ваша компания или сообщество хотят пользоваться всем потенциалом продвинутых функций и иметь максимальный доступ к моделям, то подойдет план Enterprise.

Комбинированные планы

Можно комбинировать разные типы планов на уровне организации. Подробности указаны в документации (https://oreil.ly/BZ1p6).

Независимо от выбранного плана последовательность действий остается неизменной.

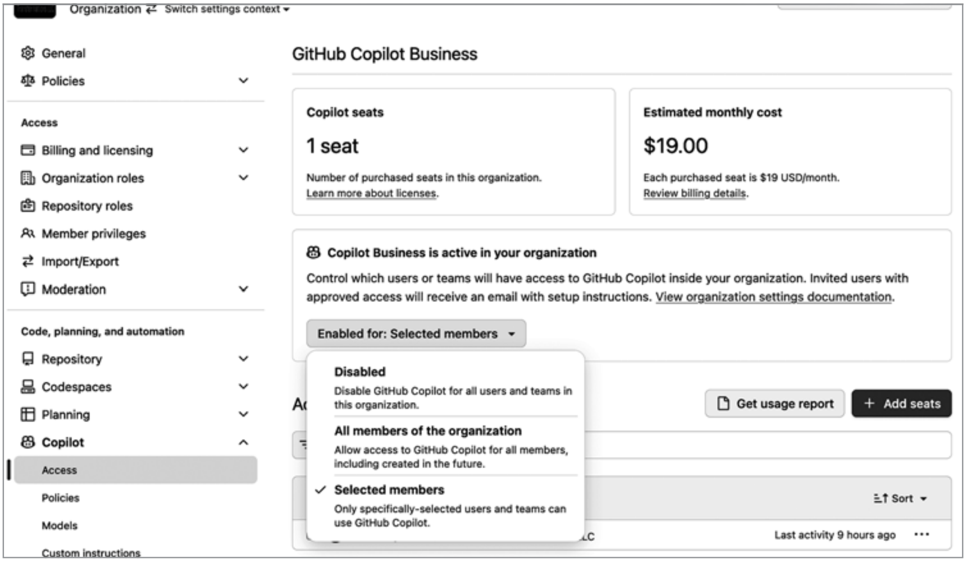
1. Введите учетные данные и зарегистрируйтесь для получения лицензии.
2. Установите способ оплаты.
3. Установите Copilot с помощью расширения или другого метода, который вы используете для добавления функциональности в IDE.

На уровне организации после настройки плана администраторы смогут добавить пользователей и управлять лицензиями (рис. 1.6).

Получив доступ к подписке Copilot, вы сможете установить его в своей IDE. В документации GitHub Copilot (https://oreil.ly/kF4rn) приведены ссылки для установки этого расширения в выбранную вами IDE. На момент написания книги поддерживаются следующие среды:

- Azure Data Studio (https://oreil.ly/uUNGn);
- Eclipse (https://oreil.ly/Mqdq2);
- JetBrains IDEs (https://oreil.ly/dlYdh);

- Vim/Neovim (https://oreil.ly/rA6oI);
- Visual Studio (https://oreil.ly/qYbh_);
- Visual Studio Code (https://oreil.ly/GYc9X);
- Xcode (https://oreil.ly/MGeCb).



GitHub Copilot Business

Copilot seats

1 seat

Number of purchased seats in this organization.
Learn more about licenses.

Estimated monthly cost

$19.00

Each purchased seat is $19 USD/month.
Review billing details.

Copilot Business is active in your organization

Control which users or teams will have access to GitHub Copilot inside your organization. Invited users with approved access will receive an email with setup instructions. View organization settings documentation.

Enabled for: Selected members

Disabled
Disable GitHub Copilot for all users and teams in this organization.

All members of the organization
Allow access to GitHub Copilot for all members, including created in the future.

✓ **Selected members**
Only specifically-selected users and teams can use GitHub Copilot.

Рис. 1.6. Управление планом Business в Copilot

GitHub Copilot Language Server

Помимо всего вышеперечисленного, GitHub предоставляет Copilot Language Server SDK (https://oreil.ly/008MF). Этот пакет позволяет интегрировать Copilot с любым редактором или IDE, использующим стандарт LSP (Language Server Protocol). Так что в будущем можно ожидать появления новых интеграций.

Резюме

Надеюсь, вы стали лучше представлять, что такое GitHub Copilot и как его использовать. Кроме того, я надеюсь, что вы заинтересовались этой темой и захотели изучить ее более подробно.

В этой главе я постарался объяснить, что такое Copilot и как он работает, перечислил ключевые критерии, которые стоит учитывать при его использовании, и описал доступные планы. Все эти объяснения рисуют картину того, как Copilot вписывается в текущие возможности и потенциал ИИ, чтобы помочь вам работать в определенной области. В данном случае ею является создание программного обеспечения.

Из этого утверждения следует несколько ключевых выводов. Во-первых, ИИ может значительно облегчить ваши усилия как программиста. Во-вторых, если вы хотите, чтобы Copilot предоставлял наилучшие результаты, то учитывайте важность контекста. И в-третьих (пожалуй, это самый важный вывод), за анализ и оценку любых предложений и ответов от ИИ в конечном счете отвечаете вы. Copilot отлично справляется со своей задачей, но лишь в пределах переданного контекста, функциональности и обучающих данных используемой им модели. И, как и в случае с программистами-людьми, эти переменные влияют на полноту, релевантность и точность (или их отсутствие) получаемых результатов.

В следующей главе вы узнаете, как использовать функциональность завершения кода в интерфейсе вашей среды разработки, что позволит вам подняться на новый уровень владения Copilot.

ГЛАВА 2

Программирование с Copilot

В предыдущей главе вы узнали, что такое Copilot и как он работает. Теперь мы можем перейти к основным принципам его использования. Предполагается, что Copilot будет работать как ИИ-помощник, поэтому вы должны понимать, как с ним взаимодействовать. По сути, все сводится к умению пользоваться интеграциями в IDE. Изучение принципов работы в этой и других главах поможет вам усвоить материал оставшейся части книги. Кроме того, вы получите ответы на следующие вопросы.

- Как обратиться с запросом к помощнику и передать необходимые указания?
- Как использовать ответы или выбрать из нескольких вариантов?
- Как общаться с помощником, чтобы получить быстрые предложения или более подробные объяснения при необходимости?
- Как наиболее эффективно использовать ИИ для упрощения задач программирования?

> **IDE = VS Code**
>
> GitHub Copilot можно установить и использовать в разных IDE, таких как VS Code, JetBrains IDEs, Eclipse и NeoVim. В одной книге мы не сможем рассмотреть все варианты, поэтому в примерах будут использоваться VS Code и GitHub Codespaces. Ознакомьтесь с документацией вашей среды, чтобы узнать, каковы особенности ее работы с Copilot.

В среде Copilot предоставляет два основных режима взаимодействия. Первый интегрируется с редактором, в нем используются контекстные меню и сопутствующие элементы управления. Для простоты я буду называть этот режим *встроенным*, так как он чаще всего используется напрямую из редактора или связанных меню. В этой главе мы обсудим основные случаи применения этого режима.

Второй режим взаимодействия осуществляется через отдельный *чат-интерфейс*, входящий в состав Copilot. Такой подход предоставляет больше возможностей для взаимодействия и получения обратной связи. Вдобавок он служит точкой доступа к более сложной функциональности Copilot, такой как *режим агента* и *расширенный режим редактирования*. Варианты использования чата мы рассмотрим в главах 3 и 4.

Скорее всего, большинство пользователей впервые начинают работать с Copilot во время взаимодействия с редактором в своей IDE. В процессе написания кода установленное расширение Copilot собирает контекст и интегрируется с удаленными компонентами, чтобы сформировать предложения по коду. Общий принцип его работы был представлен в главе 1. Пришло время рассмотреть его более подробно.

> В Copilot применяются разные сочетания клавиш для выполнения тех или иных функций. Обычно это управляющая клавиша (Command, Ctrl или Option) и буква, цифра или другой символ. Управляющая клавиша зависит от используемой операционной системы и, возможно, от клавиатуры. Вместо того чтобы каждый раз перечислять все возможные сочетания, я буду использовать обозначение Meta+<*символ*> — например, Meta+I вместо Option+I. Список сочетаний клавиш для вашей платформы можно найти в документации Copilot (https://oreil.ly/aAZ8S).

Встроенные предложения

Представим, что вы создаете файл Python для работы с простыми числами. Вы можете начать с создания файла `prime.py`. Вероятно, первым видимым признаком активности Copilot станет возможность вызова *Copilot Chat* прямо из редактора с помощью сочетания клавиш Meta+I. Это напоминание будет появляться каждый раз при редактировании любого файла, если Copilot активен и чат включен (рис. 2.1).

```
1   Press ⌘ I to ask GitHub Copilot Chat to do something. Start typing to dismiss.
```

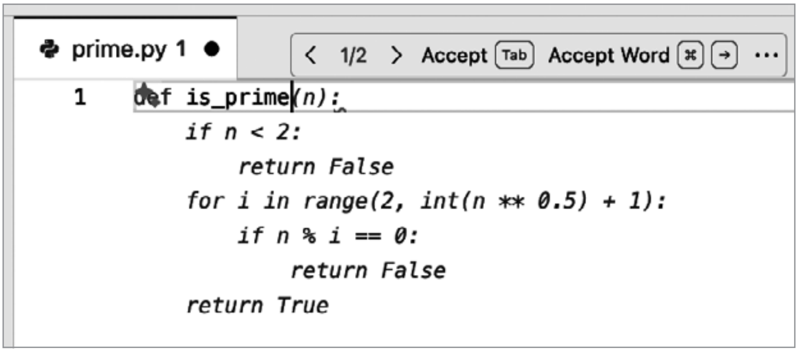
Рис. 2.1. Функция вызова Copilot Chat непосредственно в редакторе

Сам интерфейс Copilot Chat мы будем рассматривать в главе 3, а пока поговорим о более простых взаимодействиях, поскольку это поможет вам лучше понять его работу. И хотя функция вызова чата включена по умолчанию, вы можете проигнорировать ее в редакторе и начать вводить код, чтобы продолжить работу.

Чтобы вы могли увидеть пример, создадим в файле `prime.py` функцию для проверки того, является ли число простым. Начните вводить:

```
def is_prime
```

В этот момент Copilot начинает предлагать возможные варианты завершения. Например, это может быть вариант, подобный показанному на рис. 2.2. Текст, предложенный Copilot, отображается как *призрачный*. Пока предложение не принято, оно выделяется курсивом, а шрифт имеет более светлый цвет.



```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Рис. 2.2. Первое предложение

Ваши предложения могут быть другими

В этой главе, а также во всех местах книги, где показываются предложения или другие результаты работы Copilot, помните: вы взаимодействуете с генеративным ИИ, поэтому предложения, которые получите вы, могут быть совсем другими, даже если вы используете тот же контекст.

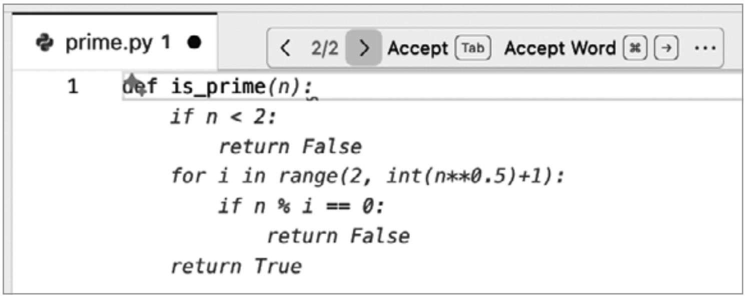
Предложения Copilot — один из возможных вариантов создания вашей функции. На рис. 2.2 справа над предлагаемым текстом есть серая панель. Она появляется, когда Copilot предлагает текст непосредственно в редакторе. На ней размещаются элементы управления для принятия всего предложения или его части и элементы, которые позволяют циклически просматривать предложения, если Copilot сгенерирует несколько вариантов.

Ваша IDE может выглядеть иначе

В книге я использую IDE в стиле VS Code. Если вы работаете в другой среде (например, от JetBrains), то интерфейсы и элементы управления в ней могут выглядеть иначе, хотя должны предоставлять сходную функциональность.

Кроме того, текст на панели напоминает о соответствующих сочетаниях клавиш и возможных действиях: принять все предложение (клавиша `Tab`), принять по одному слову (нажатием сочетания клавиш, указанного на панели и в документации) и выполнить другие операции, доступные через меню дополнительных действий, обозначенное многоточием (о нем мы поговорим позднее).

В левой части панели выводится обозначение `<1/2>`. Оно указывает, что Copilot сгенерировал два возможных предложения и сейчас вы видите вариант 1. Чтобы просмотреть другие возможные варианты, щелкайте на значках > и < на панели или нажимайте соответствующие клавиши. Например, если вы хотите просмотреть вариант 2, то щелкните на > (рис. 2.3).



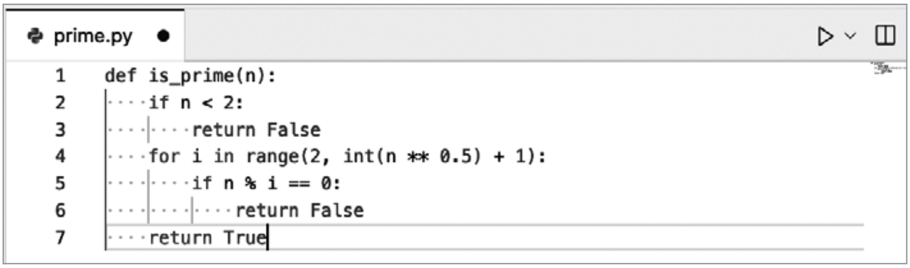
```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True
```

Рис. 2.3. Второе предложение

Чтобы принять какое-либо из предложений Copilot, перейдите к нужному варианту и нажмите клавишу Tab. Если ни один из вариантов вас не устраивает, просто продолжайте вводить код.

Варианты 1 и 2 различаются незначительно — всего лишь видами горизонтальных отступов[1]. Но Copilot представляет эти два варианта как разные. Независимо от того, какой из них вы выберете, вы получаете код, который выглядит как работоспособное тело функции, сгенерированное автоматически.

Если вы выберете вариант 1, то после того, как вы нажмете Tab, весь текст, предложенный Copilot, будет вставлен в файл. Этот текст перестает быть призрачным и выглядит точно так же, как любой другой, который вы ввели вручную (рис. 2.4).



```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Рис. 2.4. Принят вариант 1

Воспользовавшись чатом несколько раз, вы сможете предугадывать поведение интерфейса. Типичный рабочий процесс введения кода будет выглядеть так.

1. Вы принимаете или отклоняете предложение от Copilot.
2. При необходимости/желании вводите дополнительный код вручную.

---

[1] В языке программирования Python одни программисты используют для отступов пробелы, а другие — символ табуляции (Tab). Эти отступы определяют горизонтальное выравнивание кода и важны для синтаксиса Python.

3. Делаете небольшую паузу (при необходимости), чтобы Copilot успел сгенерировать варианты.
4. Copilot выдает предложения.
5. Шаги 1–4 повторяются, пока работа не будет завершена.

Полнота и *уместность* предложений Copilot могут значительно различаться. В зависимости от доступного ему контекста он может предлагать варианты завершения кода — от комментария до отдельной строки или нескольких строк, образующих готовое решение. Пока вы будете вводить код, Copilot в одних случаях может предложить вам полное определение функции, а в других — всего пару строк. Рабочий процесс, описанный выше, остается неизменным независимо от того, является ли первое предложение Copilot законченным вариантом, или же вам придется выполнить всю процедуру несколько раз, принимая предложения построчно.

Паузы для генерирования предложений

Возможно, у вас возникли вопросы по шагу 3, где упоминается пауза, во время которой Copilot генерирует варианты. В промежутках между выводом предложений происходят серьезные вычисления и обмен данными. Как упоминалось в главе 1, чтобы сгенерировать предложения, Copilot выполняет большой объем фоновой работы. Он собирает контекст, синтезирует его, пересылает по сети, передает LLM, проверяет наличие любых закономерностей, которые могут указывать на наличие уязвимостей, при необходимости проверяет эти закономерности на возможные совпадения с публичным кодом, а затем предоставляет результаты.

Все это происходит невероятно быстро. Но иногда Copilot требуется немного времени, чтобы вернуть результат после того, как вы нажмете Enter. Если вы привыкли после нажатия Return сразу нажимать Tab, чтобы принять следующие предложения Copilot, то время от времени будете опережать его. В таком случае нужно сделать небольшую паузу, прежде чем пытаться принять следующую строку. Кроме того, учтите, что иногда Copilot может генерировать пустые строки как часть вывода.

Взаимодействие с Copilot, показанное выше, имеет ряд особенностей.

- Copilot смог предложить определения функций, соответствующие контексту.
- Два варианта слегка разнились, причем различия *не были технически значимыми*, но при этом все равно представлялись как разные.
- Как Copilot получил конкретный контекст для предложений? Для этого он использовал имя файла и введенное вами имя функции.
- Если вы повторите это упражнение, то нет гарантий, что вы получите те же предложения (и даже что получите несколько предложений по коду). Генеративный ИИ не всегда возвращает одинаковые результаты, даже если контекст тот же самый.

Несколько слов о последнем пункте: встроенные предложения Copilot часто могут быть достаточно хороши для ваших задач, но иногда желательно увидеть дополнительные варианты. Для этого в Copilot предусмотрен встроенный механизм запроса *альтернативных предложений*. О нем мы и поговорим далее.

Получение альтернативных предложений по коду

Итак, если исходных предложений Copilot окажется недостаточно, то можно попросить его сгенерировать больше альтернативных вариантов. Для этого предназначено сочетание клавиш Ctrl+Enter.

Плюсы и минусы функциональности альтернативных предложений

Запросив у Copilot альтернативные предложения, можно получить больше вариантов завершения. Но у этого подхода есть минусы: на их создание и представление уйдет больше времени по сравнению со встроенными вариантами. Кроме того, альтернативный код может состоять из неполных реализаций или комментариев, что ухудшает его полезность.

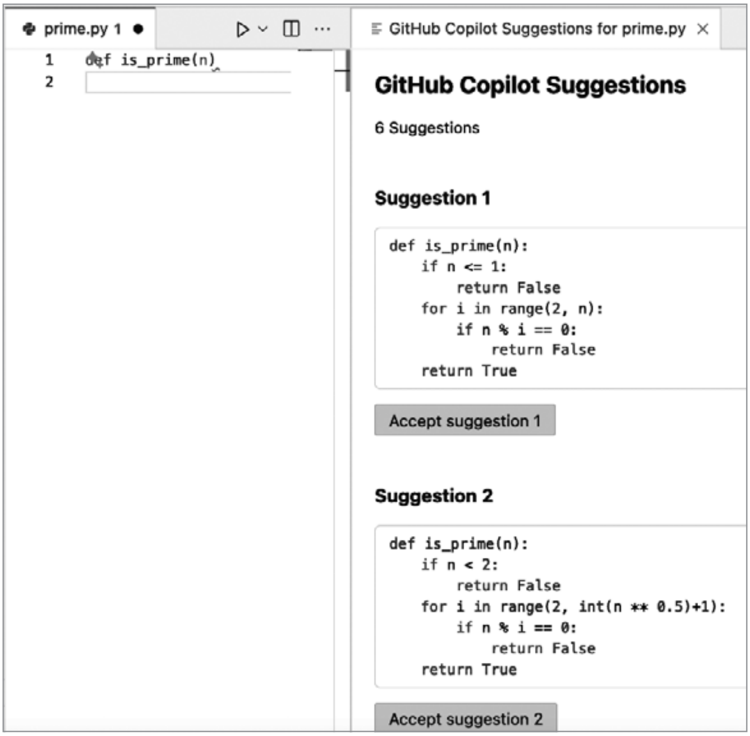
Эту функциональность лучше всего использовать перед тем, как принимать одно из встроенных предложений по коду. Вернемся к моменту, когда мы ввели имя функции, но еще не приняли предложение. В этот момент можно нажать Ctrl+Enter, и Copilot сгенерирует отдельную панель с альтернативными вариантами (рис. 2.5).

Copilot предложил шесть альтернатив, причем все они выглядят вполне нормально и различаются лишь некоторыми нюансами.

Вкладка GitHub Copilot Suggestions доступна только для чтения. Вы можете прокрутить набор предложений, чтобы просмотреть их. Если вы найдете вариант, который вам понравится, то сможете вставить его в файл, нажав кнопку Accept suggestion # (где # — номер предложения).

Если вы хотите использовать функциональность альтернативных предложений для уже написанного кода, то результаты будут различаться еще сильнее. В таком случае важно выделить код, на который Copilot должен ориентироваться при формировании своих предложений. Иначе вы можете не получить никаких результатов.

Дополнительные предложения для существующего кода будут различаться по качеству и удобству использования. На рис. 2.6 представлен набор альтернатив, сгенерированных для существующего кода.



```
def is_prime(n)
```

GitHub Copilot Suggestions

6 Suggestions

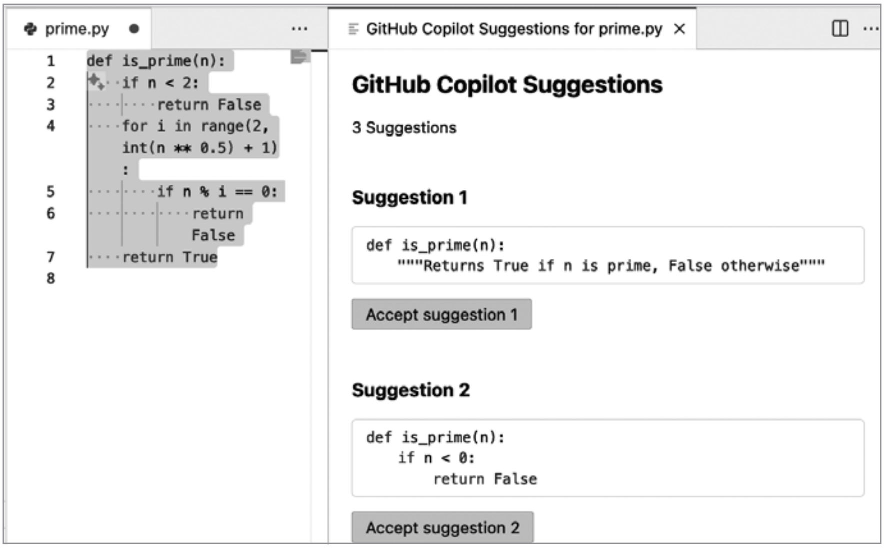
Suggestion 1

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Suggestion 2

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5)+1):
        if n % i == 0:
            return False
    return True
```

Рис. 2.5. Альтернативные предложения перед завершением



```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2,
    int(n ** 0.5) + 1)
    :
        if n % i == 0:
            return
            False
    return True
```

GitHub Copilot Suggestions

3 Suggestions

Suggestion 1

```
def is_prime(n):
    """Returns True if n is prime, False otherwise"""
```

Suggestion 2

```
def is_prime(n):
    if n < 0:
        return False
```

Рис. 2.6. Альтернативные предложения для существующего кода

Обратите внимание: на вкладке предложений всего три альтернативных варианта. Ни один из них не является полной реализацией. Таким образом, при наличии минимального контекста лучше запросить альтернативные предложения, прежде чем добавлять код.

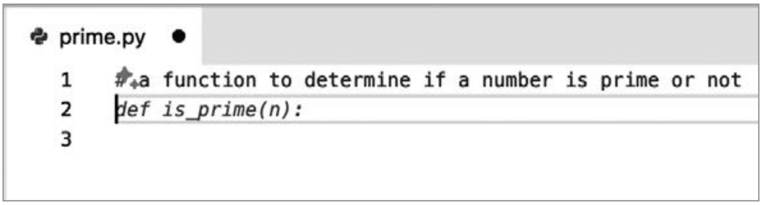
Как упоминалось в главе 1, Copilot формирует предложения, используя не только код, но и сопутствующие элементы, такие как комментарии. Обычно их добавляют только для пояснения кода, но с их помощью можно еще и управлять поведением Copilot.

Комментарии

Комментарии могут играть важную роль в контексте для Copilot. Допустим, вы вводите следующий комментарий в единственной строке нового файла:

```
# Функция проверяет, является ли число простым
```

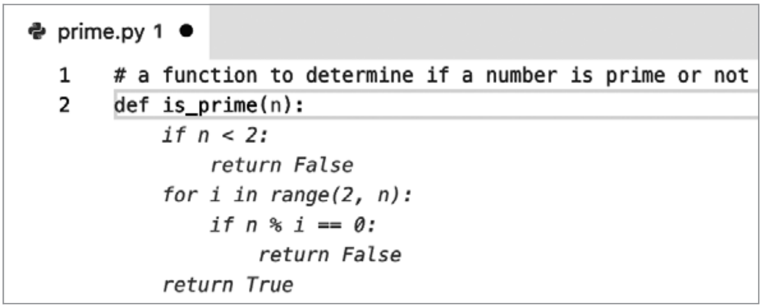
Copilot может ответить так (рис. 2.7).



```
# a function to determine if a number is prime or not
def is_prime(n):
```

Рис. 2.7. Предложение, сгенерированное на основе комментария

Если принять это предложение, то Copilot предложит остальной код тела функции (рис. 2.8).



```
# a function to determine if a number is prime or not
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Рис. 2.8. Завершение, сгенерированное на основе комментария

Таким образом, Copilot может получить из стандартных комментариев пассивный контекст. Но помимо этого, с их помощью можно указать Copilot, что он должен делать. Для этого комментарий нужно сформулировать так, чтобы он больше напоминал инструкцию.

Комментарии-инструкции

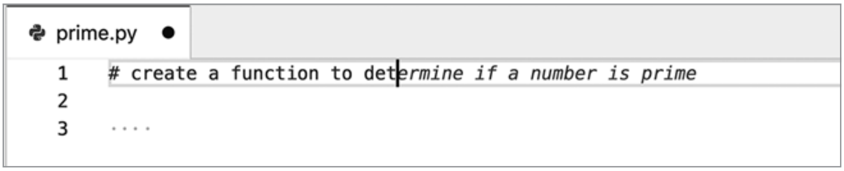
Итак, с помощью комментариев вы можете сообщить Copilot, каких действий ждете от него. Допустим, вместо заголовка функции (см. раздел «Встроенные предложения» выше):

```
def is_prime
```

вы ввели комментарий с описанием того, что должно происходить. Применительно к функции проверки простых чисел он мог бы выглядеть так:

```
# Создай функцию для проверки того, является ли число простым
```

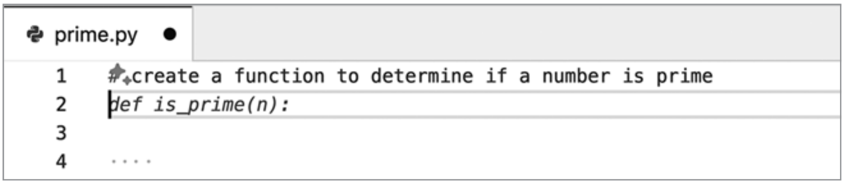
В зависимости от доступного контекста Copilot может предложить релевантное завершение вашего комментария (рис. 2.9). Оно может соответствовать вашему замыслу, а может и не соответствовать.



```
# create a function to determine if a number is prime

....
```

Рис. 2.9. Автоматическое завершение комментария

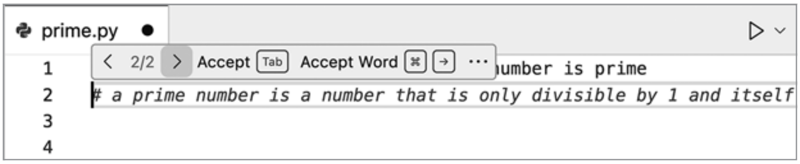
Этой строки достаточно для того, чтобы начать генерирование предлагаемого кода функции. Пример показан на рис. 2.10.



```
#create a function to determine if a number is prime
def is_prime(n):

....
```

Рис. 2.10. Сгенерированная первая строка функции

Предложенное имя функции совпадает с предыдущим, но варианты различаются. Вместо того чтобы предлагать всего одну строку, Copilot выдает несколько возможных завершений. Одно из них — заголовок функции. Второе — продолжение комментария. Эти варианты можно просмотреть на всплывающей панели или с помощью сочетаний клавиш (рис. 2.11).



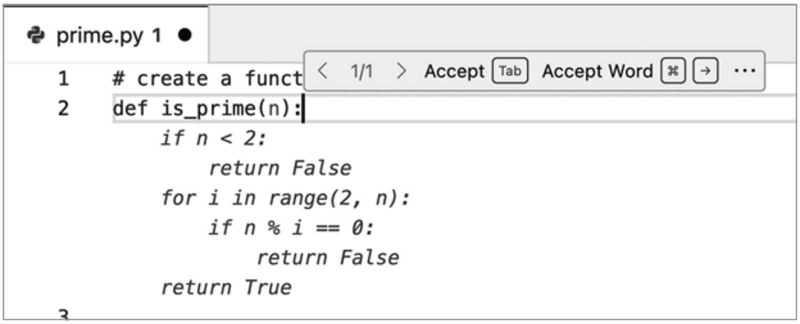
```
umber is prime
# a prime number is a number that is only divisible by 1 and itself
```

Рис. 2.11. Второй вариант — продолжение комментария

Варианты могут быть другими

В очередной раз напомню: вы работаете с генеративным ИИ, поэтому предложения, которые вы получите при использовании Copilot, могут отличаться от приведенных в книге, даже если вы возьмете тот же код примера.

На этот раз Copilot недостаточно контекста, чтобы понять, чего вы хотите: сгенерировать расширенный комментарий или начать создание кода функции, поэтому он предоставляет оба варианта. После принятия вами первого варианта Copilot получает достаточный контекст и предлагает определение функции (рис. 2.12).



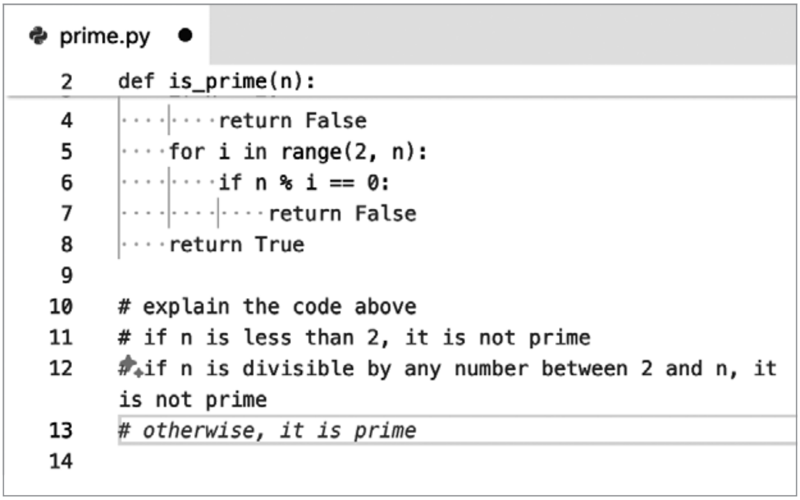
```
# create a funct
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Рис. 2.12. Полное предложение функции

Другой пример: вы можете попросить Copilot объяснить существующий код, введя такой комментарий:

```
# Объясни код, приведенный выше
```

Copilot выдает краткие закомментированные строки с описанием работы кода. На рис. 2.13 показано, что в данном случае Copilot сгенерировал три строки с комментариями (11–13) после инструкции «объясни».



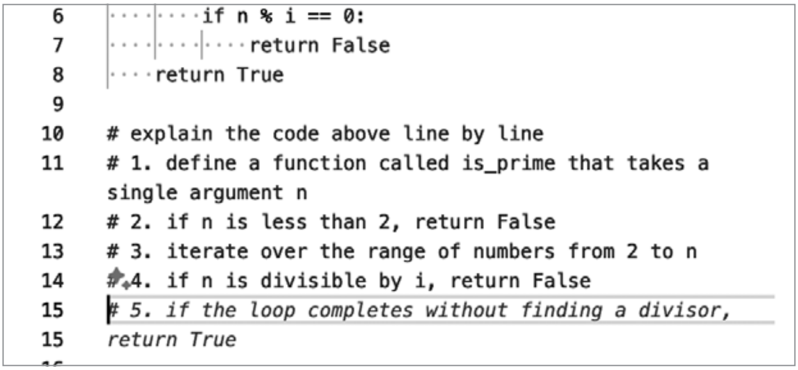
```
def is_prime(n):
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

# explain the code above
# if n is less than 2, it is not prime
# if n is divisible by any number between 2 and n, it is not prime
# otherwise, it is prime
```

Рис. 2.13. Вывод после указания Copilot объяснить код в комментариях

Кроме того, вы можете написать комментарий, чтобы получить более подробный ответ:

```
# Объясни код, приведенный выше, строку за строкой
```

Результат показан на рис. 2.14.



```
        if n % i == 0:
            return False
    return True

# explain the code above line by line
# 1. define a function called is_prime that takes a single argument n
# 2. if n is less than 2, return False
# 3. iterate over the range of numbers from 2 to n
# 4. if n is divisible by i, return False
# 5. if the loop completes without finding a divisor,
return True
```

Рис. 2.14. Ответ Copilot на уточненное задание

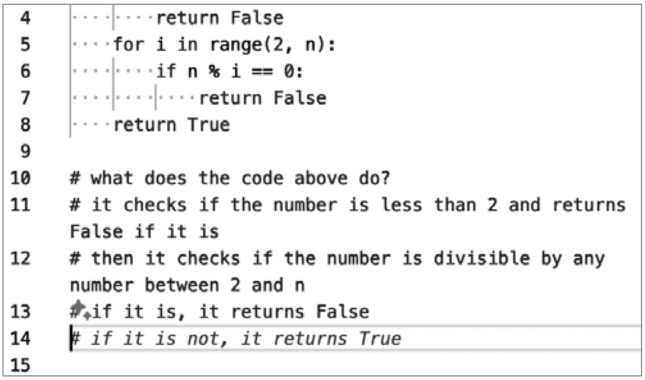
Важный момент: если вы используете комментарии для того, чтобы *дать Copilot указание сгенерировать код*, то он сделает это, основываясь на них. Для любых других инструкций, не связанных с генерацией кода (например, «объясни»), результаты будут представлены в виде *дополнительных комментариев*.

Вопросы в комментариях

Помимо выдачи указаний, комментарии можно использовать и для того, чтобы задавать вопросы Copilot. Например, вместо того, чтобы просить Copilot объяснить код, можно задать такой вопрос:

```
# Что делает код, приведенный выше?
```

Результат показан на рис. 2.15.



```
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

# what does the code above do?
# it checks if the number is less than 2 and returns False if it is
# then it checks if the number is divisible by any number between 2 and n
# if it is, it returns False
# if it is not, it returns True

```

Рис. 2.15. Использование комментария для того, чтобы задать вопрос Copilot

Комментарии можно обозначить как вопросы более формально, добавив `q:` после знака комментария, но это не обязательно. Если вы сделаете это, то Copilot выдаст ответ, начинающийся с `a:`.

Контекст вопросов и ответов

Задавать вопросы Copilot через комментарии было способом получать ответы в стиле чата до того, как Copilot Chat был добавлен в тарифные планы Copilot. Сейчас этот способ менее удобен, чем использование встроенного чат-интерфейса или главного окна чата, но все еще подходит, если вам нужны краткие ответы, добавленные непосредственно в код.

Разобравшись с тем, как взаимодействовать с Copilot напрямую из редактора, перейдем к его использованию из меню IDE.

Copilot и контекстное меню

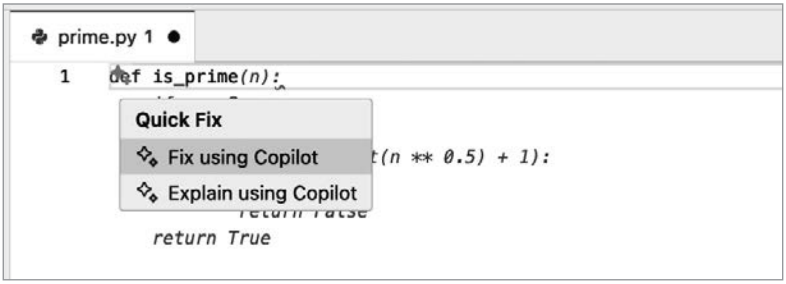
Некоторые функции Copilot вызываются из контекстных меню вашей IDE. Благодаря этой интеграции вы можете быстро запускать некоторые расширенные функции, не давая указания в чате. (О чат-интерфейсах мы поговорим в главе 3.)

Второе значение термина «контекст»

В этом разделе при описании меню термин «контекст» обозначает не информацию, которую Copilot собирает для генерирования предложений по коду, а команды Copilot, доступные в контекстных меню IDE.

Возможно, вы заметили на некоторых рисунках две маленькие *четырехконечные звездочки* в начале строк кода. Это *искра* — общепринятое обозначение ИИ (используемое не только в Copilot). В IDE этим значком обычно помечаются специальные ссылки: щелкнув на них, вы откроете меню с несколькими расширенными командами для взаимодействия с Copilot.

В примерах выше этот значок отображается там, где определение функции не завершено. Но система не может отличить функцию, которая еще только пишется, от той, в которой есть ошибка. Поэтому Copilot выдает предложение по завершению и варианты исправления ошибки. Щелкнув на значке с искрой, можно открыть контекстное меню, показанное на рис. 2.16 (возможно, оно не будет отображаться в некоторых IDE).



```
def is_prime(n):
                    t(n ** 0.5) + 1):
    return True
```

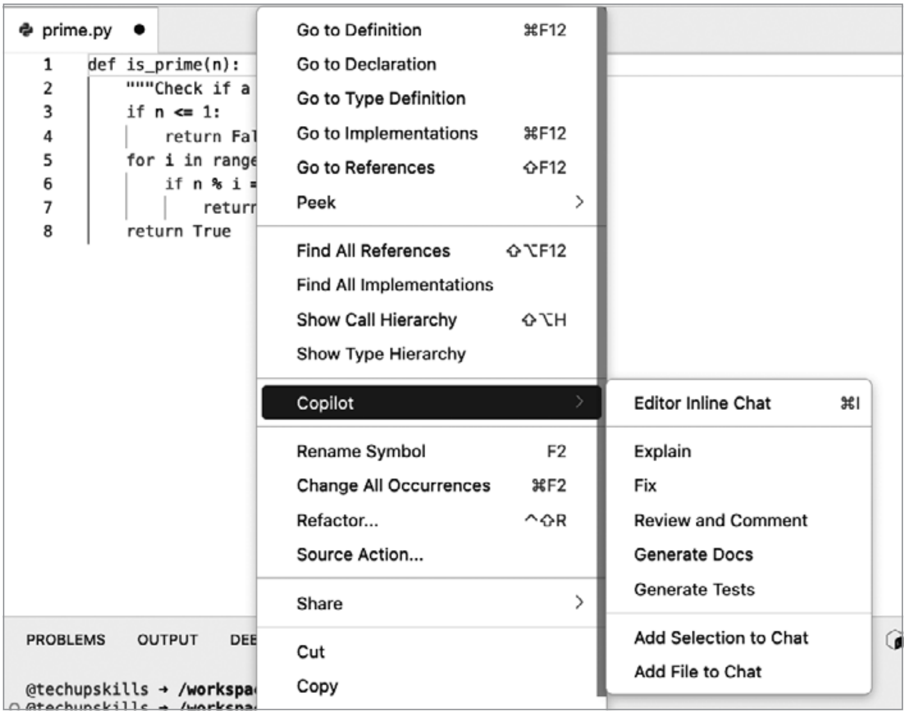
Рис. 2.16. Функция быстрого исправления с использованием Copilot

Завершение с помощью функции исправления

Иногда вместо того, чтобы получать варианты завершения кода, лучше использовать команду Fix using Copilot. Если вы выберете ее, то Copilot сгенерирует неполный код, который может привести систему в рабочее состояние. Но не стоит рассчитывать, что эта функциональность предоставит полное решение.

Когда появится больше данных для работы Copilot (например, завершенная реализация функции), в меню появятся другие команды (рис. 2.17).

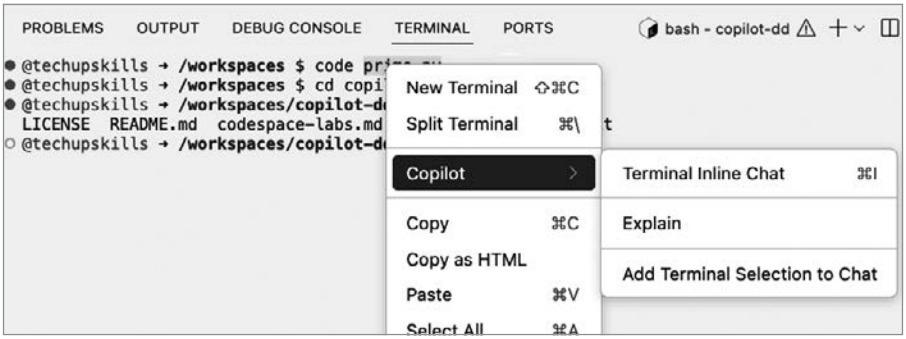
Кроме того, доступные команды меню (и их смысл) изменяются в зависимости от того, с какой частью IDE вы работаете. Например, если вы работаете в терминале, то команда меню Explain будет относиться к содержимому терминала, а не редактора. Кроме того, есть и другие команды, относящиеся исключительно к терминалу (рис. 2.18). (Это функции чата, поэтому я расскажу о них в главе 3.)



```
def is_prime(n):
    """Check if a
    if n <= 1:
        return Fal
    for i in range
        if n % i =
            return
    return True
```

```
@techupskills → /workspa
```

Рис. 2.17. Расширенные команды в контекстном меню



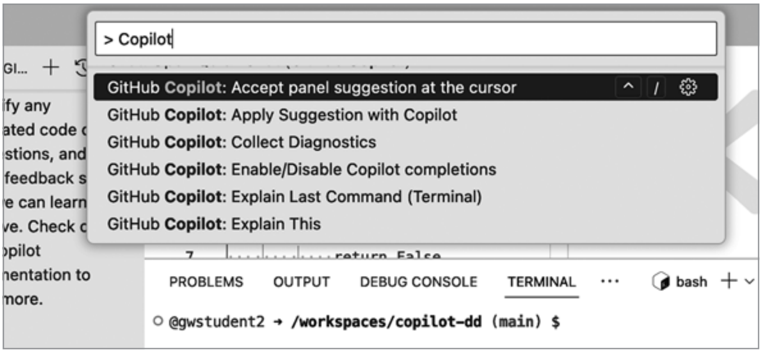
```
● @techupskills → /workspaces $ code pr
● @techupskills → /workspaces $ cd copi
● @techupskills → /workspaces/copilot-d
  LICENSE  README.md  codespace-labs.md
○ @techupskills → /workspaces/copilot-d
```

Рис. 2.18. Команды Copilot в терминале

Таким образом, в контекстных меню доступна функциональность Copilot, которая может быть полезна в зависимости от задачи, стоящей перед разработчиком. Кроме того, в VS Code и аналогичных IDE можно выполнять команды, относящиеся к Copilot, используя интерфейс *палитры команд*, как описано в следующей врезке.

ДОСТУП К COPILOT ЧЕРЕЗ ПАЛИТРУ КОМАНД

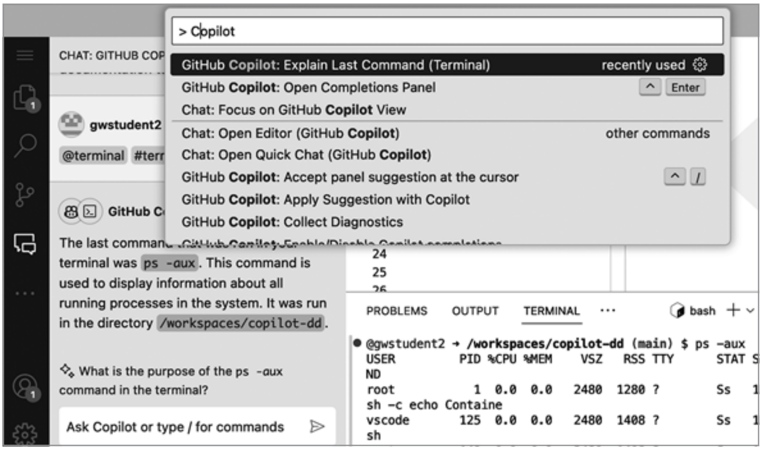
В интегрированных средах в стиле VS Code есть интерфейс для быстрого вызова команд, относящихся к установленным расширениям. Он называется *палитрой команд*. Чтобы открыть ее, нажмите Meta+P или F1. В появившемся диалоговом окне можно выбрать или ввести название расширения, команды, символа или файла. Например, если у вас установлено расширение Copilot, то вы можете получить доступ к связанным с ним командам, введя в палитре команд слово *Copilot* (рис. 2.19).



```
○ @gwstudent2 → /workspaces/copilot-dd (main) $
```

Рис. 2.19. Обращение к функциональности Copilot из палитры команд

Элементы в этом списке можно выполнять как команды. На рис. 2.20 показано выполнение функции Copilot «Объясни» (Explain).



The last comman terminal was `ps -aux`. This command is used to display information about all running processes in the system. It was run in the directory `/workspaces/copilot-dd`.

What is the purpose of the ps -aux command in the terminal?

```
● @gwstudent2 → /workspaces/copilot-dd (main) $ ps -aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT S
ND
root          1  0.0  0.0   2480  1280 ?        Ss   1
sh -c echo Containe
vscode      125  0.0  0.0   2480  1408 ?        Ss   1
sh
```

Рис. 2.20. Пример использования палитры команд

Помимо доступа к *стандартным* функциям, упоминавшимся выше (например, объяснение и исправление кода), с помощью контекстного меню можно поручить Copilot выполнение другой важной задачи: ревью кода и предоставление обратной связи о нем. О том, как это сделать, поговорим далее.

Использование Copilot для ревью кода

Copilot может стать рецензентом вашего кода. На GitHub вы можете добавить Copilot в качестве рецензента для pull-запросов (см. главу 9). В IDE можно попросить Copilot выполнить ревью локального кода и предоставить обратную связь. Это может быть особенно полезным для выявления проблем до отправки кода в общее хранилище. Для проверки можно использовать стандартное поведение Copilot или же задать способ выполнения ревью.

Использование стандартных средств ревью Copilot

Допустим, у вас есть следующий блок кода, который должен реализовать простой класс, представляющий точку на плоскости:

```
class Point:
    def __init__(self, x, y, z=0): self.x = x; self.y = y
    def distance(self, other): return (self.x - other.x)**2 + \
    (self.y - other.y)**2 + (self.z - other.z)**2**0.5
p1 = Point(1, 2); p2 = Point(4, 6, 0)
print(p1.distance(p2))
```

Если вы программируете на Python, то, возможно, заметите некоторые проблемы с этим кодом. Предложим Copilot выполнить его ревью. Выделите код в редакторе, щелкните правой кнопкой мыши и выберите команду **Copilot ▸ Review and Comment** (рис. 2.21).

Copilot обработает запрос, и если обнаружит проблемы или готов порекомендовать возможные действия (в том числе такие простые, как добавление документации), то во встроенном режиме добавит в код комментарии и предлагаемые изменения. На рис. 2.22 приведен пример ревью кода в Copilot.

В диалоговом окне обратной связи есть значки со стрелками, используемые для перехода к другим комментариям, а также элементы для просмотра обратной связи в чате, закрытия диалогового окна и его сворачивания. В предлагаемых изменениях текущий код выделяется красным цветом (со знаком минус), а новый — зеленым (со знаком плюс). Нажимая кнопки в нижней части диалогового окна, можно применить изменения или отказаться от них, а затем перейти к следующей группе предложений.

Помимо поведения по умолчанию, в настройках можно указать, на что Copilot должен обращать внимание во время проверки.

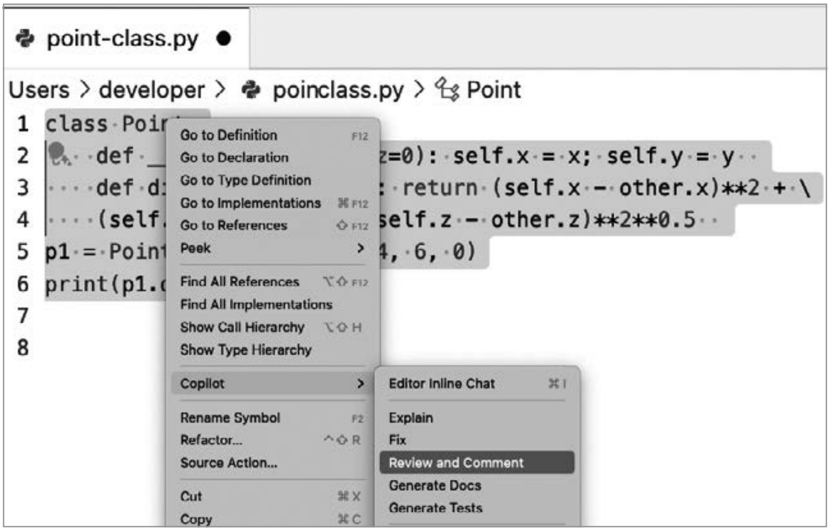
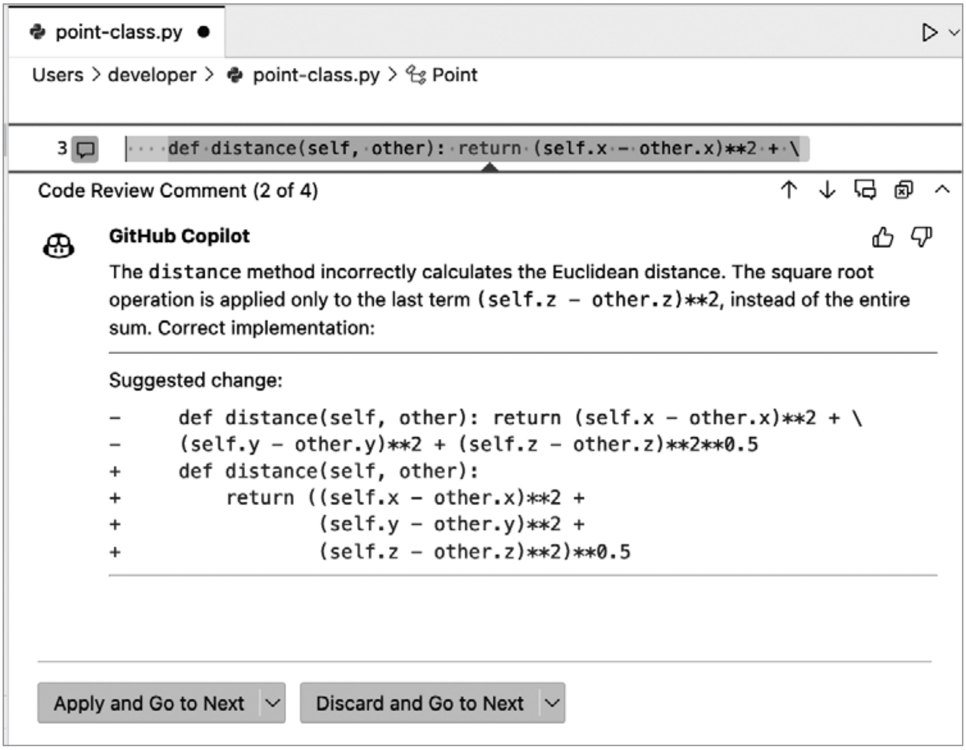


Рис. 2.21. Выбор функции ревью кода из меню



```
3 ···def·distance(self,·other):·return·(self.x·-·other.x)**2·+·\
```

GitHub Copilot

The `distance` method incorrectly calculates the Euclidean distance. The square root operation is applied only to the last term `(self.z - other.z)**2`, instead of the entire sum. Correct implementation:

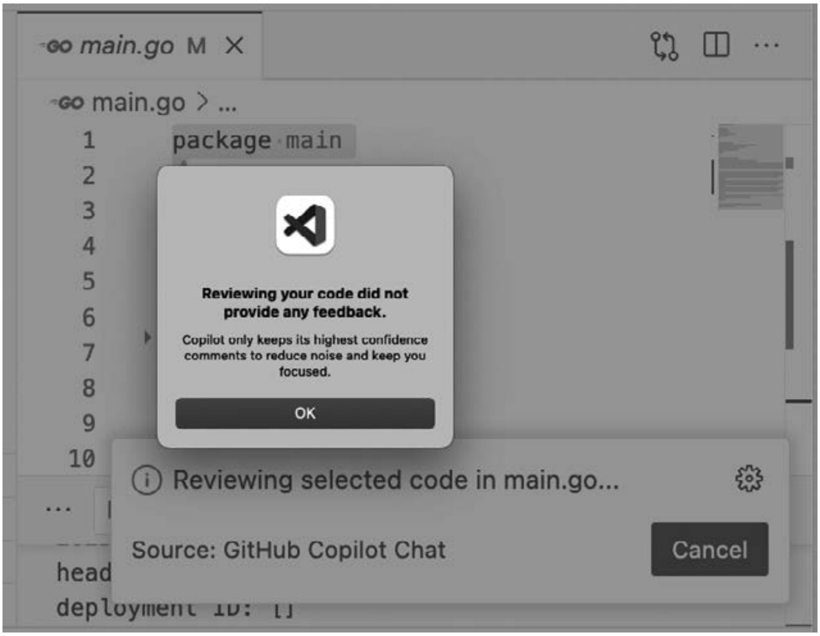
Suggested change:

```
-     def distance(self, other): return (self.x - other.x)**2 + \
-     (self.y - other.y)**2 + (self.z - other.z)**2**0.5
+     def distance(self, other):
+         return ((self.x - other.x)**2 +
+                 (self.y - other.y)**2 +
+                 (self.z - other.z)**2)**0.5
```

Рис. 2.22. Обратная связь при ревью в Copilot

Создание инструкций для нестандартного ревью

По умолчанию Copilot сообщает только о тех проблемах, которые считает важными. Из-за этого он нередко сообщает об отсутствии замечаний по вашему коду, если обнаружил в нем элементы, которые могли бы пометить другие проверяющие, например не соответствующие лучшим практикам или рекомендациям по стилю программирования. На рис. 2.23 показан пример сообщения, которое вы увидите, если Copilot не обнаружит никаких проблем.



```
package main
```

Reviewing your code did not provide any feedback.

Copilot only keeps its highest confidence comments to reduce noise and keep you focused.

Reviewing selected code in main.go...

Source: GitHub Copilot Chat

```
head
deployment ID: []
```

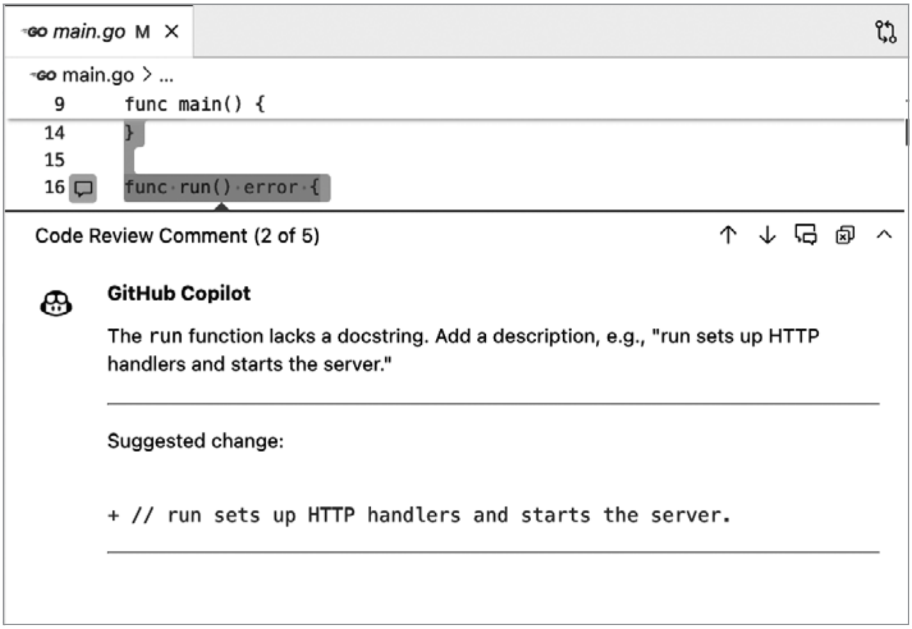
Рис. 2.23. Отсутствие замечаний при проверке, выполняемой Copilot

Если вам потребуется более подробная обратная связь, то можете предоставить набор инструкций, которыми должен руководствоваться Copilot. Это можно сделать двумя способами.

Во-первых, вы можете отредактировать файл `settings.json` и добавить собственные инструкции в раздел `github.copilot.chat.codeReview.instructions`. Например, правило могло бы выглядеть так:

```
"github.copilot.chat.reviewSelection.instructions": [
    {
        "text": "Ensure all functions have proper docstrings"
    }
]
```

Добавив правило в файл настроек, можно попросить Copilot выполнить ревью кода снова. На этот раз в выводе будет помечено условие, которое вы добавили в файле настроек (рис. 2.24).



```
 9    func main() {
14    }
15
16    func run() error {
```

Code Review Comment (2 of 5)

GitHub Copilot

The run function lacks a docstring. Add a description, e.g., "run sets up HTTP handlers and starts the server."

Suggested change:

```
+ // run sets up HTTP handlers and starts the server.
```

Рис. 2.24. Сообщение о проблеме в файле настроек

При необходимости можно добавить свое правило (и до пяти дополнительных) в виде Markdown-разметки в локальный файл, сохранить его по конкретному пути `.github/copilot-review-guidelines.md`, и Copilot автоматически обнаружит его и будет использовать.

Если вы предпочитаете сохранить инструкции в другом файле, то создайте его и добавьте ссылку на него в файл `settings.json`, используя ключ `file:`. Это может выглядеть примерно так:

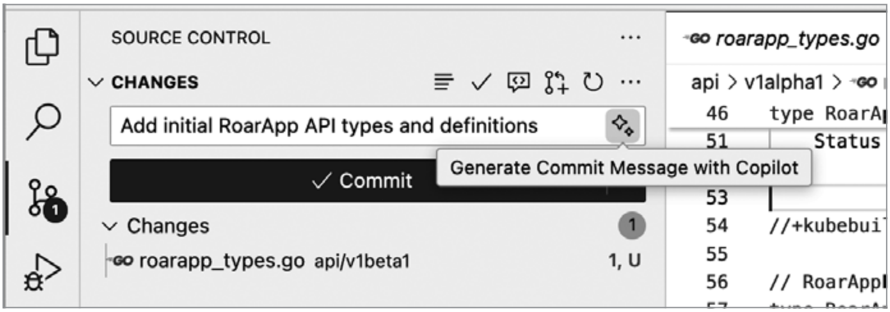
```
"github.copilot.chat.reviewSelection.instructions": [
    {
       "file": "./docs/review-guide.md"
    }
 ]
```

Предполагается, что в кодовой базе есть локальный файл `review-guide.md`, сохраненный в каталоге docs и содержащий следующий текст:

```
- Убедиться в том, что все функции имеют правильные doc-строки.
```

ГЕНЕРИРОВАНИЕ АВТОМАТИЧЕСКИХ СООБЩЕНИЙ О КОММИТАХ

Выполнив ревью кода, Copilot может помочь зафиксировать изменения, когда вы будете готовы сделать это. Как показано на рис. 2.25, если вы используете панель SOURCE CONTROL, то вместо того, чтобы самим вводить сообщение о коммите, вы можете попросить Copilot предложить вам такое сообщение, основанное на изменениях. Для этого щелкните на значке искры в конце поля с сообщением о коммите.



Add initial RoarApp API types and definitions

Рис. 2.25. Запрос на генерирование сообщения о коммите у Copilot

Вы даже можете дать нестандартные[1] инструкции для генерации сообщения о коммите. Подробнее об этом — в документации VS Code.

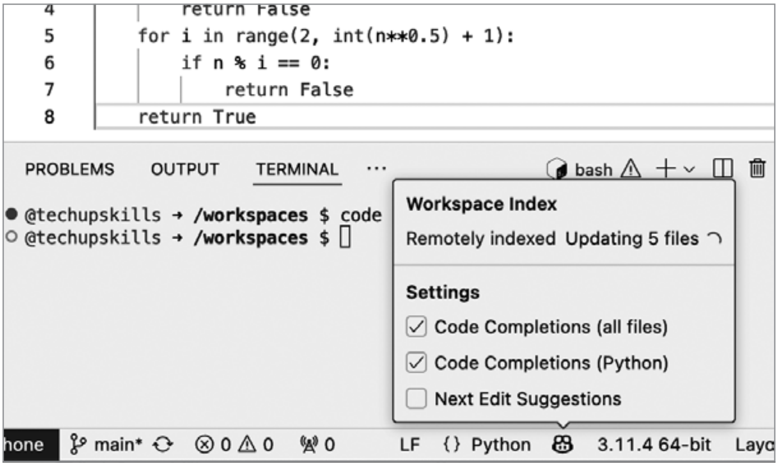
Чтобы завершить наше обсуждение программирования с Copilot в IDE, кратко рассмотрим дополнительные элементы управления и параметры конфигурации, которые помогут вам настраивать варианты взаимодействия с Copilot в соответствии с вашими потребностями.

Определение конфигурации Copilot в IDE

Установив расширение Copilot (и выполнив аутентификацию, если это необходимо), вы сможете обращаться в VS Code к функциям Copilot, используя элементы управления. Один из них находится в строке состояния в нижней части IDE. Щелкнув на нем, вы сможете отключить функцию выдачи предложений по завершению кода для всех файлов или только для тех, с которыми вы работаете в данный момент. Например, это могут быть файлы Python (рис. 2.26).

---

[1] Нестандартные инструкции — инструкции, которые прописаны самим пользователем. Их также называют кастомными (custom).



```
    return False
for i in range(2, int(n**0.5) + 1):
    if n % i == 0:
        return False
return True
```

```
● @techupskills → /workspaces $ code
○ @techupskills → /workspaces $ ▯
```

Workspace Index

Remotely indexed Updating 5 files

Settings

☑ Code Completions (all files)

☑ Code Completions (Python)

☐ Next Edit Suggestions

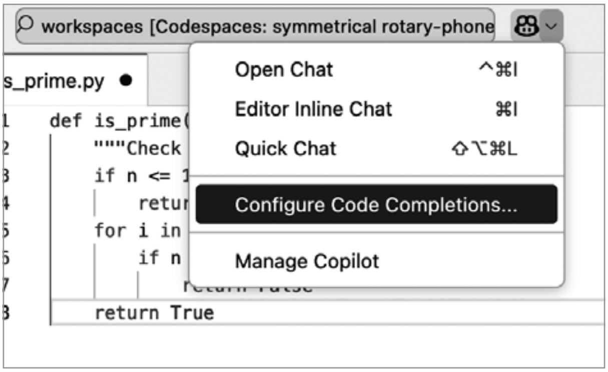
Рис. 2.26. Нижний элемент управления Copilot

Зачем отключать функцию завершения кода

Если завершать код — одна из основных задач Copilot, то зачем отключать эту функцию? Дело в том, что предложения иногда начинают раздражать или мешать. Например, если вы пытаетесь быстро ввести большой объем кода или уже знаете, как писать код, то, возможно, не захотите видеть всплывающие предложения Copilot.

Вариант **Next Edit Suggestions** в диалоговом окне относится к автоматизированному *пакетному редактированию*, рассматриваемому в главе 4.

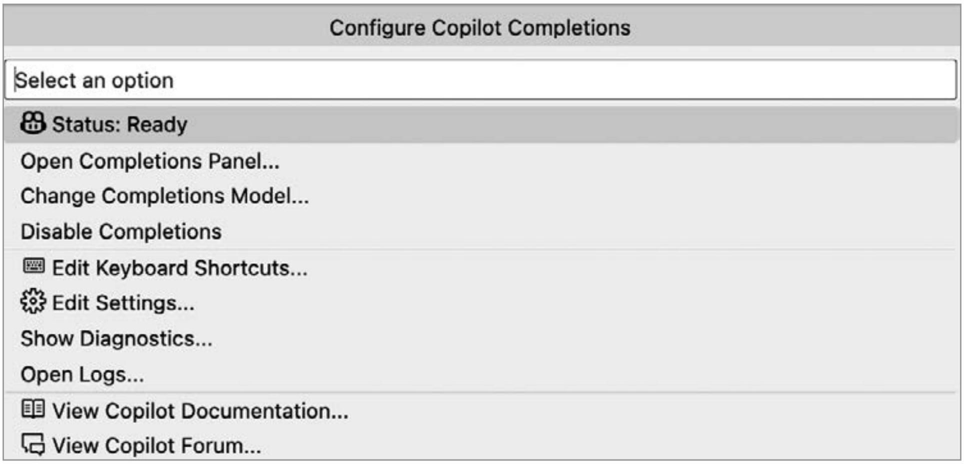
В верхней части IDE находится еще один значок Copilot. Он позволяет открывать различные чат-интерфейсы, настраивать параметры завершения кода и управлять настройками Copilot в GitHub (рис. 2.27).



```
def is_prime(
    """Check
    if n <= 1
        retur
    for i in
        if n
    return True
```

Рис. 2.27. Верхний элемент управления Copilot

Настройки чата мы рассмотрим в главе 3. Выбрав команду Manage Copilot, вы перейдете на страницу настроек (https://oreil.ly/OlrYN) вашей учетной записи Copilot на GitHub (если вы ввели свои учетные данные). Выбрав команду Configure Code Completions, вы откроете набор параметров для настройки завершения кода (рис. 2.28).



Configure Copilot Completions

Select an option

Status: Ready
Open Completions Panel...
Change Completions Model...
Disable Completions
Edit Keyboard Shortcuts...
Edit Settings...
Show Diagnostics...
Open Logs...
View Copilot Documentation...
View Copilot Forum...

Рис. 2.28. Настройка завершения кода

Назначение одних параметров очевидно, другие же могут быть не совсем понятны. Приведу краткие описания трех из них.

Open Completions Panel

Открывает список альтернативных вариантов завершения для кода, активного в данный момент. Это та же функциональность, которую мы уже рассматривали ранее (доступная при нажатии Ctrl+Enter).

Change Completions Model

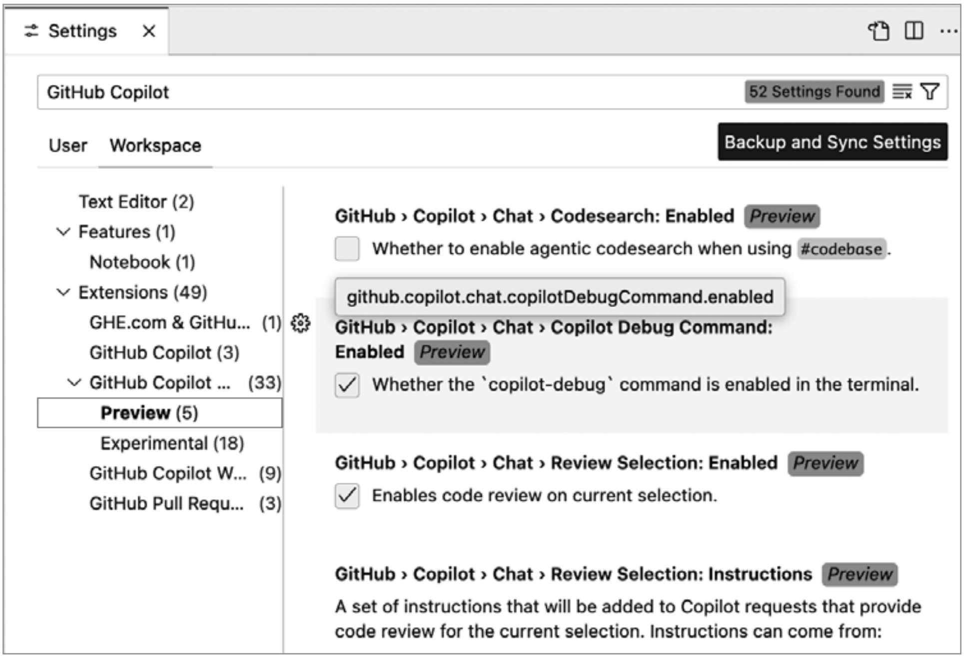
Позволяет выбрать другую модель ИИ (отличную от модели по умолчанию), если та доступна. Обратите внимание: это касается только модели, используемой для завершения кода, но не модели для чатов.

Disable Completions

Как и команда, доступная при выборе нижнего элемента управления Copilot (см. рис. 2.26), позволяет включать и отключать предложения по завершению кода.

Задать другие параметры конфигурации Copilot можно с помощью команды Edit Settings. Если выбрать ее, то в IDE открывается окно настроек со строкой поиска GitHub Copilot. Того же можно добиться, если щелкнуть на значке с шестеренкой в основном интерфейсе IDE и начать вводить в строку поиска `GitHub Copilot`, чтобы найти соответствующие настройки.

На рис. 2.29 показан экран Settings для элементов, соответствующих запросу GitHub Copilot. Обратите внимание на подкатегории слева: редактор, функции и группу расширений. В этой группе находятся настройки для всех расширений, связанных с GitHub Copilot.



GitHub › Copilot › Chat › Codesearch: Enabled *Preview*

☐ Whether to enable agentic codesearch when using `#codebase`.

github.copilot.chat.copilotDebugCommand.enabled

GitHub › Copilot › Chat › Copilot Debug Command: Enabled *Preview*

☑ Whether the `copilot-debug` command is enabled in the terminal.

GitHub › Copilot › Chat › Review Selection: Enabled *Preview*

☑ Enables code review on current selection.

GitHub › Copilot › Chat › Review Selection: Instructions *Preview*

A set of instructions that will be added to Copilot requests that provide code review for the current selection. Instructions can come from:

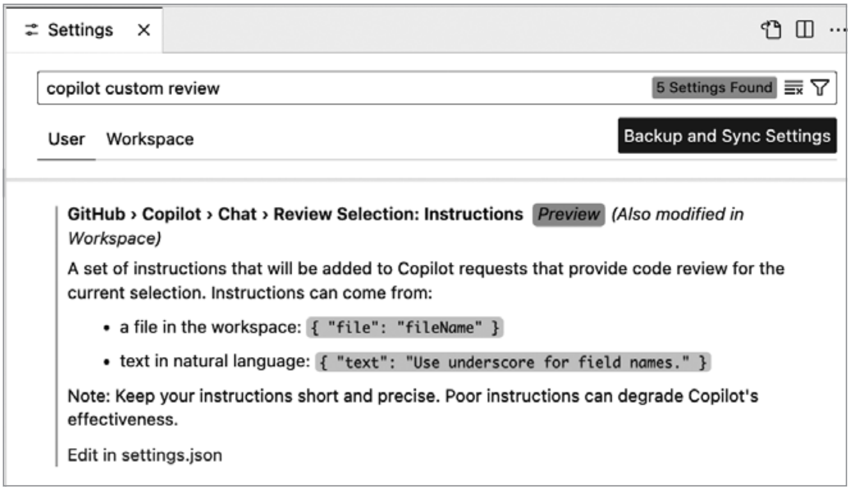
Рис. 2.29. Настройки GitHub Copilot в IDE

Здесь можно задать настройки *уровня пользователя* и *рабочего пространства* (по отдельности). Вкладки наверху слева позволяют переключаться между этими категориями.

В некоторых разделах настроек вы найдете параметры или функции с пометками Experimental или Preview. Это значит, что они еще не выпущены в эксплуатацию. *Предварительные* (preview) функции ближе к официальному выпуску и предназначены для публичного тестирования и сбора отзывов от пользователей, которые соглашаются использовать такие функции. *Экспериментальные* (experimental) же представляют собой ранние концепции разработки, которыми вы можете пользоваться на свой страх и риск. Перед выпуском в эксплуатацию их либо дорабатывают, либо удаляют, либо значительно изменяют.

РЕДАКТИРОВАНИЕ ФАЙЛОВ ЧЕРЕЗ SETTINGS (ВКЛАДКА НАСТРОЕК)

Если вы найдете параметр конфигурации, который можно сохранить в файле (например, `settings.json`), то обычно в окне отображается ссылка для изменения файла. Например, если вы захотите добавить собственные правила ревью кода (о чем мы говорили ранее в этой главе) и найдете соответствующий параметр, то в нижней части окна будет отображаться ссылка для прямого редактирования файла `settings.json` (рис. 2.30).



GitHub › Copilot › Chat › Review Selection: Instructions Preview *(Also modified in Workspace)*

A set of instructions that will be added to Copilot requests that provide code review for the current selection. Instructions can come from:

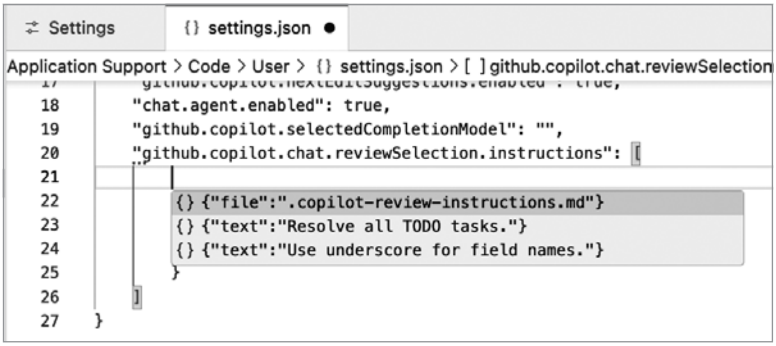
- a file in the workspace: `{ "file": "fileName" }`
- text in natural language: `{ "text": "Use underscore for field names." }`

Note: Keep your instructions short and precise. Poor instructions can degrade Copilot's effectiveness.

Edit in settings.json

Рис. 2.30. Обращение к файлу из окна Settings

Щелчок на ссылке открывает указанный файл, чтобы вы могли отредактировать его напрямую, используя исходные предложения по содержимому (рис. 2.31).



```
    "chat.agent.enabled": true,
    "github.copilot.selectedCompletionModel": "",
    "github.copilot.chat.reviewSelection.instructions": [
        
        {} {"file":".copilot-review-instructions.md"}
        {} {"text":"Resolve all TODO tasks."}
        {} {"text":"Use underscore for field names."}
        }
    ]
}
```

Рис. 2.31. Редактирование файла, открытого из окна Settings

Если вы используете другую IDE, то в ней тоже есть похожий способ вызова и редактирования настроек Copilot. Я рекомендую следить за блогами GitHub о Copilot (https://oreil.ly/qdhTv), чтобы быть в курсе последних обновлений или изменений настроек. Кроме того, можно периодически изучать доступные параметры. Иногда вы будете находить дополнительные настройки поведения Copilot, о которых могли не знать.

Резюме

В этой главе мы рассмотрели наиболее простые способы взаимодействия с GitHub Copilot напрямую из среды разработки во время написания кода. Copilot оперативно предоставляет предложения в редакторе по мере того, как вы вводите код. Их нужно проверить на полноту и адекватность. Не все предложения одинаково полезны, но они могут сэкономить вам время и/или ускорить работу над кодом и упростить его понимание.

Качество предложений в основном зависит от объема контекста, доступного для Copilot в текущей среде: имя файла, существующий код, другие файлы, комментарии и локальный индекс. Кроме того, с помощью комментариев можно описать задачу, которую для вас должен решить Copilot, и задать вопросы, на которые он отвечает комментариями.

Вдобавок Copilot может выполнять ревью кода, находить недочеты и предлагать возможные улучшения. Процесс ревью можно настроить с помощью набора пользовательских инструкций.

Copilot предоставляет немало полезных возможностей, но иногда нужно настроить его поведение или включить/отключить определенную функцию. Некоторые изменения можно внести с помощью элементов управления Copilot, имеющихся в IDE. Либо же в настройках среды можно найти отдельные параметры конфигурации, в том числе экспериментальные или находящиеся на стадии предварительного ознакомления.

Итак, вы узнали, как работать с предложениями Copilot непосредственно в редакторе. Далее мы перейдем к наиболее гибкому способу взаимодействия с Copilot и его базовой моделью: через чат-интерфейс. Этой теме посвящена глава 3.

ГЛАВА 3

Взаимодействие с Copilot в IDE

На популярных ИИ-платформах, таких как ChatGPT, основным каналом взаимодействия с LLM является чат-интерфейс. Как и при общении с человеком, непрерывный диалог с ИИ воспринимается как более естественный. Вдобавок он позволяет более подробно изучить интересующие вас темы.

Встроенные чат-интерфейсы Copilot предоставляют те же преимущества, но ориентированы на область программирования. Благодаря функциональности, рассмотренной в главе 2, вы можете быстро получить от Copilot конкретные предложения по коду. Они основаны на текущем контексте вашего кода. Однако возможность свободно и в интерактивной форме беседовать о вопросах программирования может быть такой же полезной, как и обсуждение таких проблем или идей с коллегами-людьми. Фактически вскоре становится трудно представить использование Copilot без функций чата.

В этой главе вы узнаете, как использовать чат Copilot (Copilot Chat, или просто чат) и взаимодействовать с ним. Мы рассмотрим как базовые, так и расширенные возможности чата.

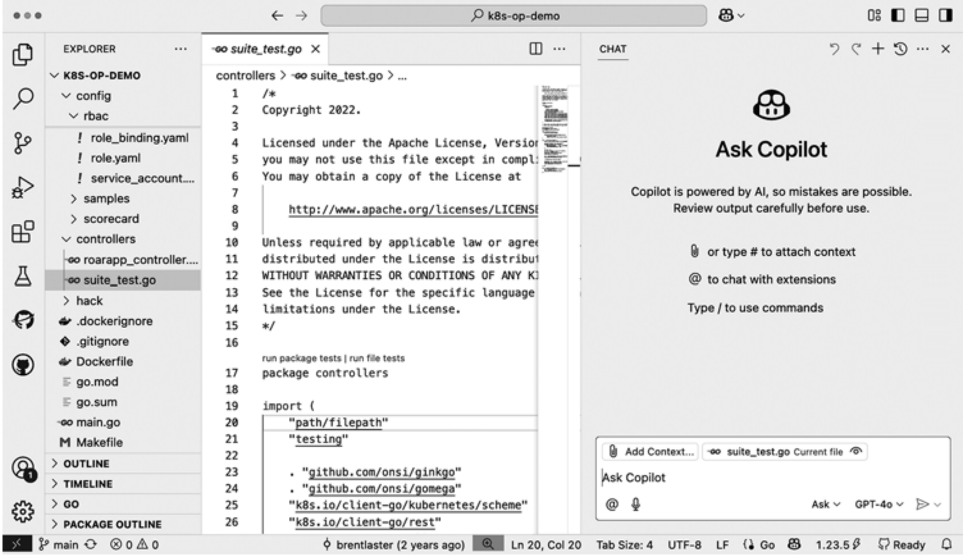
Сначала мы разберем, как вызвать основной интерфейс чата, анализировать его вывод и управлять несколькими сеансами чата.

После этого мы кратко обсудим, как лучше всего формулировать промпты для интерфейсов чата, а затем изучим сами эти интерфейсы. Чтобы вы могли использовать промпты наиболее эффективно, мы рассмотрим различные комбинации клавиш и модификаторы, встроенные в Copilot, такие как участники и чат-переменные. Кроме того, вы узнаете, как использовать чат в терминале вашей IDE.

Наконец, я покажу, как создать для чата пользовательские инструкции, которые Copilot должен использовать для генерирования ответов, и как лучше всего справляться с галлюцинациями чата и его некорректными ответами. Начнем с того, как быстро начать диалог с Copilot.

Доступ к главному интерфейсу чата

Когда вы устанавливаете Copilot в своей IDE, он поставляется вместе с Copilot Chat. Основной его интерфейс представляет собой панель в правой части экрана, отделенную от редактора (рис. 3.1). Вдобавок чат-интерфейс предоставляет больше места на экране для ввода, вывода и ведения диалогов.



```
/*
Copyright 2022.

Licensed under the Apache License, Versio
you may not use this file except in compl
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENS

Unless required by applicable law or agre
distributed under the License is distribu
WITHOUT WARRANTIES OR CONDITIONS OF ANY K
See the License for the specific language
limitations under the License.
*/

package controllers

import (
    "path/filepath"
    "testing"

    . "github.com/onsi/ginkgo"
    . "github.com/onsi/gomega"
    "k8s.io/client-go/kubernetes/scheme"
    "k8s.io/client-go/rest"
```

Ask Copilot

Copilot is powered by AI, so mistakes are possible. Review output carefully before use.

or type # to attach context

@ to chat with extensions

Type / to use commands

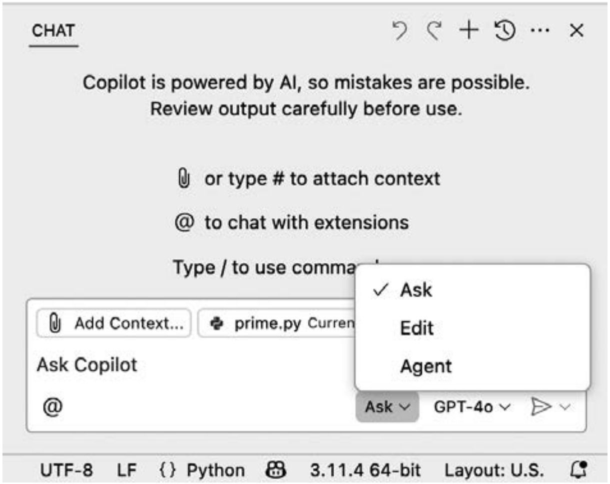
Рис. 3.1. Чат-интерфейс Copilot *(справа)*

Чат-интерфейс в GitHub

В этой главе рассматривается использование чат-интерфейса в VS Code. Чат-интерфейс Copilot широко используется и в GitHub (https://github.com/copilot). Более подробно об этом мы поговорим в главе 9.

Copilot Chat предлагает несколько режимов взаимодействия с ИИ. Режим по умолчанию, позволяющий вести диалог, называется *режимом вопросов* (Ask). Кроме того, через чат-интерфейс в Copilot можно получить доступ к расширенным возможностям редактирования и некоторым автономным функциям разработки в режиме агента. Между режимами можно переключаться (рис. 3.2).

В этой главе мы сосредоточимся на основных принципах использования чата в режиме вопросов, а о возможностях редактирования и режиме агента поговорим в главе 4.



Copilot is powered by AI, so mistakes are possible.
Review output carefully before use.

Рис. 3.2. Выбор режима чата

В интеграции с IDE в верхней строке находится значок Copilot. При щелчке на нем можно открыть различные чат-интерфейсы в среде. В частности, можно впервые или повторно открыть главный чат-интерфейс, если тот не виден, — для этого следует выбрать верхний пункт списка (рис. 3.3).

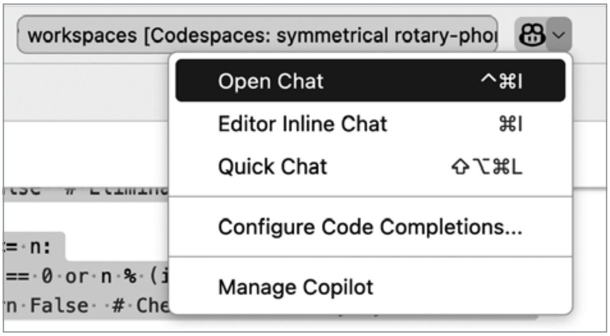


Рис. 3.3. Открытие элементов управления чатом

В начале нового диалога с чат-интерфейсом вы получите сообщение: «Copilot работает на основе ИИ, поэтому возможны ошибки. Внимательно проверьте ответ, прежде чем использовать его».

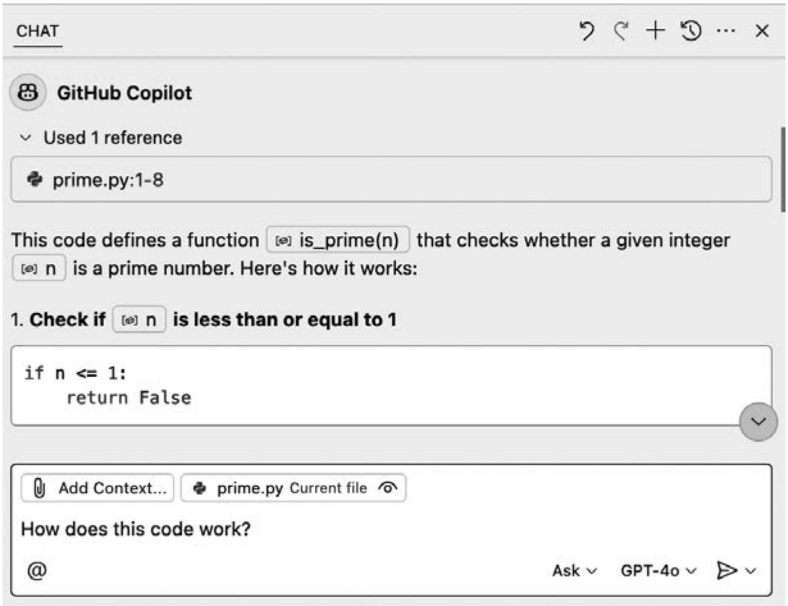
Предупреждение о возможных неточностях в ответах ИИ Copilot относится ко всем возвращаемым им ответам, а не только к тем, которые получены в чат-интерфейсе. Как упоминалось в главе 1, необходимо проверять все ответы, сгенерированные Copilot.

Параметры интерфейсов встроенного чата в редакторе (Editor Inline Chat) и быстрого чата (Quick Chat) будут рассмотрены позднее в этой главе.

В главном чат-интерфейсе можно ввести запрос или промпт в текстовом окне Ask Copilot. Допустим, вы выделили код `is_prime` из примеров главы 2 и хотите, чтобы Copilot объяснил, как он работает. Простой запрос может выглядеть так:

```
Как работает этот код?
```

Обработав запрос, Copilot выдает ответ; пример показан на рис. 3.4.



This code defines a function is_prime(n) that checks whether a given integer n is a prime number. Here's how it works:

1. **Check if n is less than or equal to 1**

```
if n <= 1:
    return False
```

How does this code work?

Рис. 3.4. Результат для запроса в чате

Обратите внимание: ответ представлен в более разговорном стиле. Он состоит из полноценных предложений, и его можно прочитать и понять, даже не имея опыта программирования на Python. Кроме того, в ответе есть примеры кода, дополняющие объяснение.

Сценарии использования функции объяснения

На первый взгляд неясно, зачем просить Copilot объяснить созданный вами код. Но представьте, что он объясняет код, который написан кем-то другим и который вы еще не понимаете. Это может быть код, написанный на незнакомом вам языке. Кроме того, можно попросить Copilot объяснить сложный алгоритм. Такое объяснение при дальнейшей доработке может даже стать основой для псевдокода.

Ответы в этом интерфейсе содержат много полезной информации, в том числе такие элементы, которые не являются непосредственной частью объяснения, но связаны с ним. Далее мы поговорим о полном наборе информации в выводе.

Анализ вывода в чате

Помимо текста в чате, в выводе стоит обратить внимание на несколько других элементов. Чтобы изучить их, мы используем уже знакомый нам пример.

Прямо под заголовком GitHub Copilot выводится сообщение: Used 1 reference (Используется 1 ссылка). В данном случае под *ссылкой* (reference) подразумевается контент из проекта (или связанный с ним), который, по мнению Copilot, был релевантным для создания ответа. Чаще всего это текущий активный файл в редакторе, но могут добавляться и другие файлы, зависимости, пользовательские настройки или иные типы контента, которые не являются файлами. (Например, это могут быть команды, если вы используете промпт со ссылкой на терминал.)

На рис. 3.4 под логотипом GitHub Copilot я развернул раздел ссылок, чтобы увидеть контент, учитывавшийся при генерировании ответа. В этом случае ссылка указывает на код из единственного активного файла `prime.py`. Это контекст по умолчанию, который также отображается в верхней части области ввода текста, где файл обозначен как текущий (Current file) (и поэтому является контекстом по умолчанию). Более подробно о контексте для промптов мы поговорим в разделе «Эффективное использование диалогового окна Prompt» далее в этой главе.

После раздела со ссылками следует полный текст ответа от Copilot (в формате Markdown), полученный во время нашего взаимодействия с ним. (Еще раз напомню: мы используем генеративный ИИ, поэтому вы можете получить разные ответы даже на абсолютно идентичные промпты.)

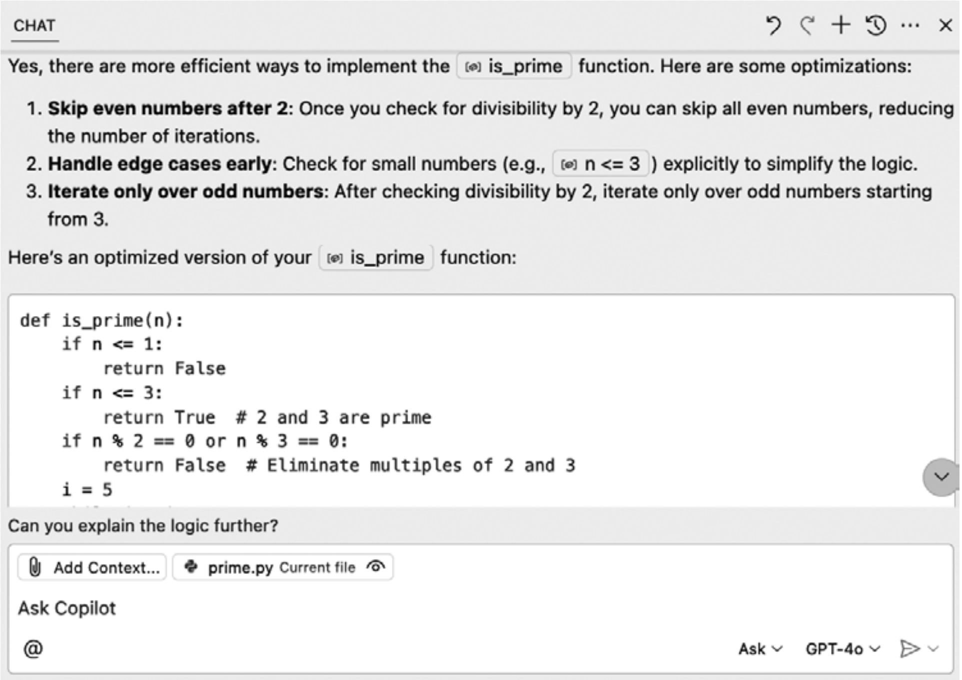
~~~
Этот код определяет функцию `is_prime(n), которая проверяет, является
ли заданное целое число `n` простым. Вот как он работает:

### 1. **Проверить, что `n` меньше либо равно 1**
```python
if n <= 1:
    return False
```
- Числа, меньшие или равные 1, не являются простыми, поэтому функция
немедленно возвращает `False`.

### 2. **Перебрать потенциальные делители**
```python
~~~

~~~
for i in range(2, int(n**0.5) + 1):
    if n % i == 0:
        return False
```
- Функция перебирает целые числа `i`, начиная с 2 до квадратного корня
из `n` (включительно). Квадратный корень вычисляется с помощью выражения
`n**0.5`, а выражение `int(n**0.5) + 1` гарантирует, что диапазон
содержит целую часть квадратного корня.
- Если `n` делится на любое `i` в диапазоне (`n % i == 0`),
 это не простое число, а функция возвращает `False`.

### 3. **Вернуть `True`, если делители не найдены**
```python
return True
```
- Если цикл завершается и никакие делители не найдены, то функция
делает вывод, что `n` является простым числом, и возвращает `True`.

### **Ключевые моменты**
- **Оптимизация**: цикл проверяет делители только до квадратного корня из `n`,
 что сокращает количество итераций по сравнению с проверкой
 всех чисел вплоть до `n-1`.
- **Определение простого числа**: простым называется число больше единицы,
 которое делится только на 1 и само себя. Данная функция эффективно
 реализует это определение.

### Пример
- Ввод: `is_prime(7)`
  - Цикл проверяет делители 2 и 3 (так как `int(7**0.5) + 1 = 3`).
  - 7 не делится на 2 и 3, поэтому функция возвращает `True`.
- Ввод: `is_prime(9)`
  - Цикл проверяет делители 2 и 3.
  - 9 делится на 3, поэтому функция возвращает `False`.
~~~

Ответ содержит подробное объяснение действий кода в виде текста (законченные предложения, написанные так, словно Copilot говорит с вами), а также фрагменты кода из файла. Это одно из ключевых преимуществ работы с чат-интерфейсом: текст разговора содержит код. В результате образуется полноценный диалог, подобный тому, который мог бы возникнуть при общении с коллегой.

Разговор легко продолжить в чате. Например, вы можете дать ссылку на свой код и спросить Copilot, нет ли более эффективного способа реализовать эту логику. На рис. 3.5 показан ответ Copilot с оптимизированной версией кода. Снова обратите внимание на текстовое пояснение, сопровождающее пример кода.

Вывод в чат-интерфейсе доступен только для чтения. Но если навести указатель на блок кода, то Copilot покажет всплывающую панель со значками для разных способов переноса кода из окна в редактор вашей IDE. На рис. 3.6 показана такая панель с расширенными параметрами.



Yes, there are more efficient ways to implement the `is_prime` function. Here are some optimizations:

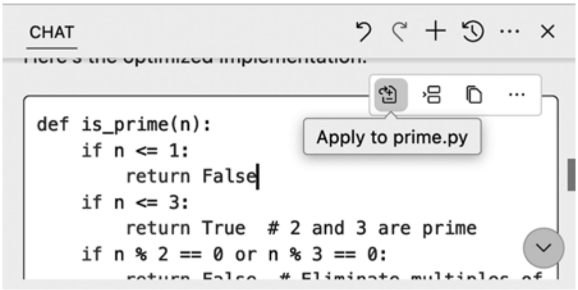
1. **Skip even numbers after 2**: Once you check for divisibility by 2, you can skip all even numbers, reducing the number of iterations.
2. **Handle edge cases early**: Check for small numbers (e.g., `n <= 3`) explicitly to simplify the logic.
3. **Iterate only over odd numbers**: After checking divisibility by 2, iterate only over odd numbers starting from 3.

Here's an optimized version of your `is_prime` function:

```
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True  # 2 and 3 are prime
    if n % 2 == 0 or n % 3 == 0:
        return False  # Eliminate multiples of 2 and 3
    i = 5
```

Can you explain the logic further?

Рис. 3.5. Вопрос к Copilot о возможности более эффективной реализации кода



```
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True  # 2 and 3 are prime
    if n % 2 == 0 or n % 3 == 0:
```

Рис. 3.6. Значки для переноса кода из чат-интерфейса в редактор

Вы можете выполнить следующие действия (соответствуют значкам слева направо):

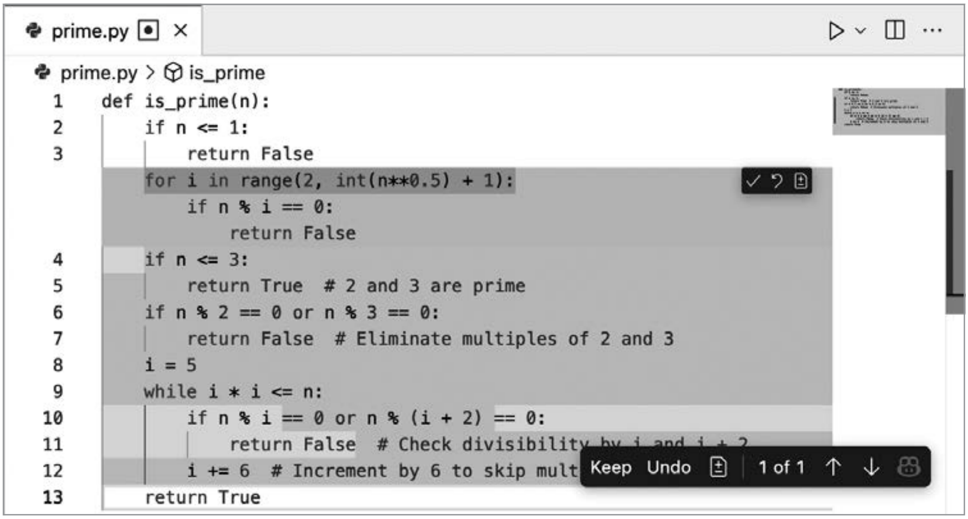
- применить изменения к текущему активному файлу для ревью;
- вставить текст в позицию курсора в текущем активном файле;
- скопировать текст в буфер обмена;
- вставить текст в новый файл;
- вставить текст в терминал.

Настройка меню наведения указателя мыши

По умолчанию для получения доступа к двум последним командам в списке (вставка в новый файл и вставка в терминал) необходимо навести указатель мыши на многоточие (...) на всплывающей панели, выбрать вариант More Actions, а затем выбрать нужный вариант.

Но если навести указатель мыши на один из значков и щелкнуть правой кнопкой мыши, то откроется меню, в котором можно выбрать, какие значки будут отображаться на панели при наведении указателя. Если установить все флажки в списке, то на всплывающей панели будут показаны все значки и вам не придется использовать интерфейс для дополнительных действий (...).

Почти все команды интуитивно понятны, но команда Apply заслуживает отдельного пояснения. Если щелкнуть на ней, то Copilot вносит предложенные им изменения в файл в редакторе, показывая их непосредственно в коде, а вы можете сохранить (Keep) или отменить (Undo) каждое изменение (рис. 3.7).



```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    if n <= 3:
        return True  # 2 and 3 are prime
    if n % 2 == 0 or n % 3 == 0:
        return False  # Eliminate multiples of 2 and 3
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False  # Check divisibility by i and i + 2
        i += 6  # Increment by 6 to skip mult
    return True
```

Рис. 3.7. Применение изменений напрямую из чата

При выборе команды Insert at Cursor весь выделенный код в сеансе редактора заменится кодом из окна. Для примера представьте, что выделили приведенный выше код, а затем выбрали эту команду. В таком случае ваша предыдущая реализация функции `is_prime` будет заменена реализацией из чата.

Помимо тех элементов управления, которые появляются при наведении указателя мыши на сгенерированный код на панели чата, в верхней части панели и в области ввода текста всегда остаются доступными другие элементы управления. Посмотрим, что они позволяют делать, и начнем с тех, которые помогают управлять сеансами чата.

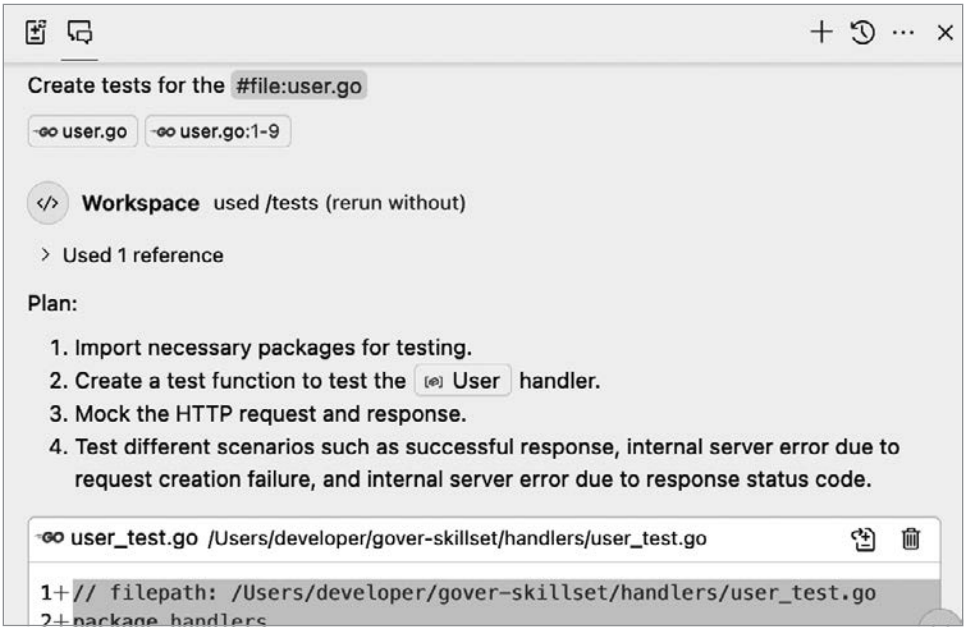
Управление сеансами чата

В верхней строке панели чата находятся элементы для управления сеансами чата (рис. 3.8).

Рис. 3.8. Верхние значки в чат-интерфейсе

Значки со стрелками влево и вправо позволяют перемещаться вперед и назад между сеансами. Большой знак + позволяет запустить новый сеанс чата вместо того, чтобы продолжать текущий. Запуск нового сеанса полезен, если вам нужно обсудить новую тему, но вы не хотите, чтобы Copilot извлек контекст из предыдущих обсуждений в существующий чат.

В одном сеансе чата Copilot отслеживает историю общения, в том числе серию промптов. Например, на рис. 3.9 я попросил его сгенерировать тесты для одного из моих файлов.


Create tests for the #file:user.go

Workspace used /tests (rerun without)

Plan:

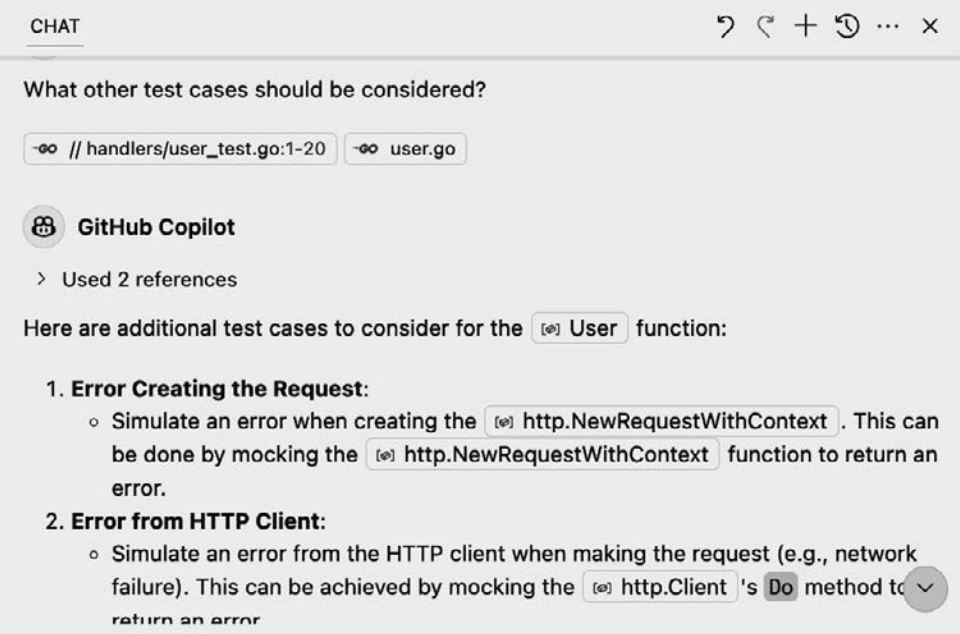
1. Import necessary packages for testing.
2. Create a test function to test the User handler.
3. Mock the HTTP request and response.
4. Test different scenarios such as successful response, internal server error due to request creation failure, and internal server error due to response status code.

user_test.go /Users/developer/gover-skillset/handlers/user_test.go

```
// filepath: /Users/developer/gover-skillset/handlers/user_test.go
package handlers
```

Рис. 3.9. Создание тестов для файла

Затем я задал более общий вопрос о том, какие еще сценарии использования следует рассмотреть. Copilot дал ответ, актуальный для контекста и продолжающий тему предыдущего вопроса (рис. 3.10).



What other test cases should be considered?

// handlers/user_test.go:1-20 · user.go

GitHub Copilot

Used 2 references

Here are additional test cases to consider for the `User` function:

1. **Error Creating the Request**:
   - Simulate an error when creating the `http.NewRequestWithContext`. This can be done by mocking the `http.NewRequestWithContext` function to return an error.
2. **Error from HTTP Client**:
   - Simulate an error from the HTTP client when making the request (e.g., network failure). This can be achieved by mocking the `http.Client`'s `Do` method to return an error

Рис. 3.10. Последующие вопросы

Щелкнув на значке Show Chats (часы с круговой стрелкой), можно вывести список чатов, с которыми вы работали в диалоговом окне (рис. 3.11). Щелкнув на любом пункте списка, вы перейдете к этому чату.

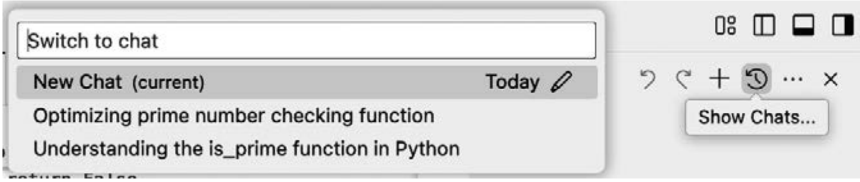
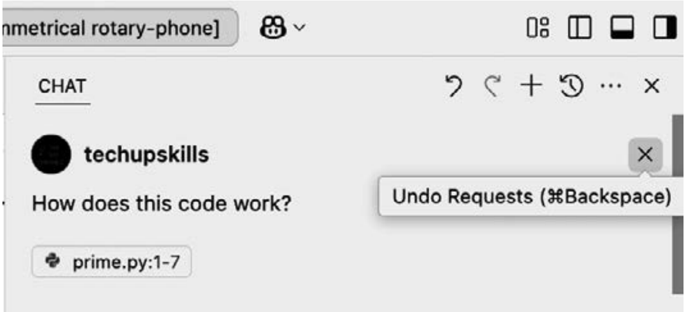


Рис. 3.11. Отображение списка чатов

Щелкнув на элементе управления Views and More Actions (значок с многоточием), вы получаете доступ к дополнительным командам. В частности, можете открыть текущий чат в отдельном сеансе редактора или окне, отправить разработчикам отзыв о работе чата, а также перейти к последним замечаниям по релизу.

При щелчке на значке X в верхней части окно чата закроется. Кроме того, можно удалить отдельный чат — наведите на него указатель мыши и нажмите X в правом верхнем углу чата (рис. 3.12). Название функции сформулировано немного странно (Undo Requests), но в настоящее время она удаляет чат. Это действие иногда может пригодиться для удаления обсуждений, которые перестали быть актуальными.



How does this code work?

Рис. 3.12. Удаление чата

Обратите внимание: содержимое сеанса чата можно экспортировать или сохранить. Об этом мы поговорим в главе 6.

Итак, сеансами чата можно управлять, используя специальные элементы. Мы взаимодействуем с Copilot в чате через промпты, поэтому стоит поговорить о том, как их создавать.

Промпт-инжиниринг для Copilot Chat

Промпт-инжиниринг для GitHub Copilot Chat — это процесс написания четких, конкретных инструкций, которые помогут ИИ сгенерировать полезные и точные ответы. Промпт для Copilot Chat может быть простым вопросом, быстрой командой (см. далее в этой главе) или конкретным указанием, связанным с вашим проектом. Чтобы предоставить релевантные ответы, Copilot не только использует ваш промпт, но и анализирует контекст из открытых файлов и историю чата.

Если вы работаете над более масштабной задачей, то можете начать с широкого описания цели и подготовить почву для ее достижения. Например, можете начать с формулировки: «Напиши функцию, которая проверяет, является ли число простым». Указав общую цель, можно дополнить ее конкретными требованиями или ограничениями, например: «Функция должна возвращать `true` для простых чисел или выдавать ошибку, если входное значение не является положительным целым числом». Постепенный переход от общего к частному помогает Copilot понять как задачу в целом, так и детали, что приводит к выдаче более точных ответов.

Еще один эффективный метод — предоставление примеров: ввод и ожидаемый вывод, модульные тесты. Так, если вы хотите, чтобы функция извлекала даты из строки, то предоставьте строки и ожидаемый массив дат — это поможет Copilot сгенерировать код, соответствующий вашим потребностям.

Кроме того, Copilot Chat поддерживает такие расширенные возможности, как обращение к конкретным файлам, использование чат-переменных и интеграция с участниками и расширения для задач, связанных с конкретной предметной областью. (О чат-переменных и участниках мы поговорим далее в этой главе, а о расширениях Copilot — в главе 10.) Вы можете выделять код, использовать ключевые слова или перетаскивать файлы в чат, чтобы предоставить Copilot больше контекста. Главное здесь — итеративный подход: если первоначальный ответ не совсем правильный, то уточняйте промпт или контекст, пока не получите нужный результат.

Рассматривая промпт-инжиниринг как интерактивный диалог, вы сможете использовать Copilot Chat для написания кода, отладки ошибок, генерирования тестов и выдачи ответов на вопросы по проекту. Чем четче вы доносите свою цель, тем лучше Copilot сможет помочь в процессе разработки.

Дополнительная информация

Информацию о промпт-инжиниринге можно найти в документации Copilot (https://oreil.ly/QnqjI).

Когда вы вводите промпты, в поле для текста отображаются дополнительные элементы для настройки среды и источников.

Эффективное использование диалогового окна Prompt

В диалоговом окне чата есть элементы управления, используемые при отправке промптов или запросов (рис. 3.13).

Эти элементы управления используются для выполнения следующих операций:

- добавления элементов в контекст;
- выбора участника чата для помощи в обработке подсказки;
- определения действий Copilot после отправки подсказки;
- выбора модели ИИ;
- выбора режима использования чат-интерфейса (Ask, Edit, Agent).

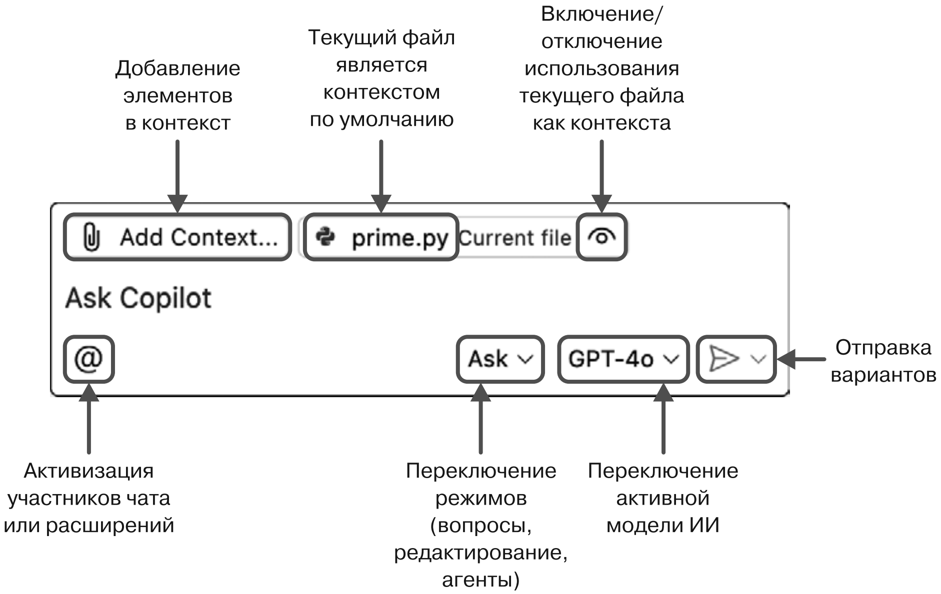


Рис. 3.13. Элементы управления в диалоговом окне Prompt

Как упоминалось выше, в этой главе мы сосредоточимся на режиме Ask (режим вопросов), а Edit и Agent рассмотрим в главе 4, так что последний пункт перечня подробно описываться не будет. Остальные же операции описаны в подразделах ниже.

Добавление элементов в контекст

Контекст в данном случае понимается как совокупность релевантных элементов, являющихся частью проекта, которые Copilot может использовать для понимания задачи и формулировки ответов. На простейшем уровне, если у вас в редакторе есть открытый активный файл, он будет использоваться в качестве текущего контекста. На рис. 3.13 такой файл обозначается как prime.py Current file. Если вы захотите исключить его из текущего контекста, то сделайте активным другой файл или щелкните на значке с глазом, чтобы отменить использование текущего файла в качестве контекста.

Чтобы вручную добавить другие элементы вашего проекта в качестве контекста, щелкните на значке со скрепкой и надписью Add Context. На экране появится список дополнительных элементов: содержимое редактора, отдельные файлы, символы, текст из терминала и даже снимки экрана. Есть и иные варианты для поиска другого контента, в частности веб-страниц или примеров использования функций, переменных и символов. На рис. 3.14 показан неполный список элементов, которые можно добавить с помощью функции Add Context.

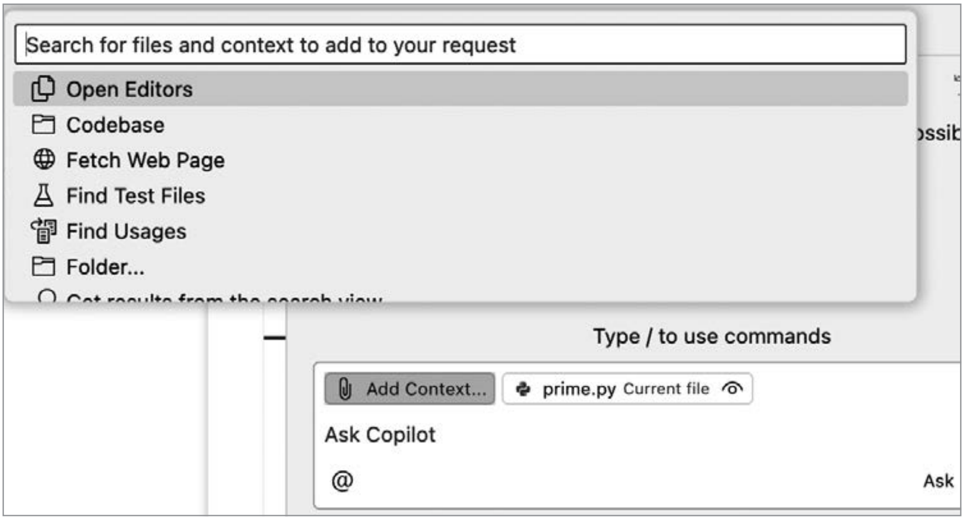
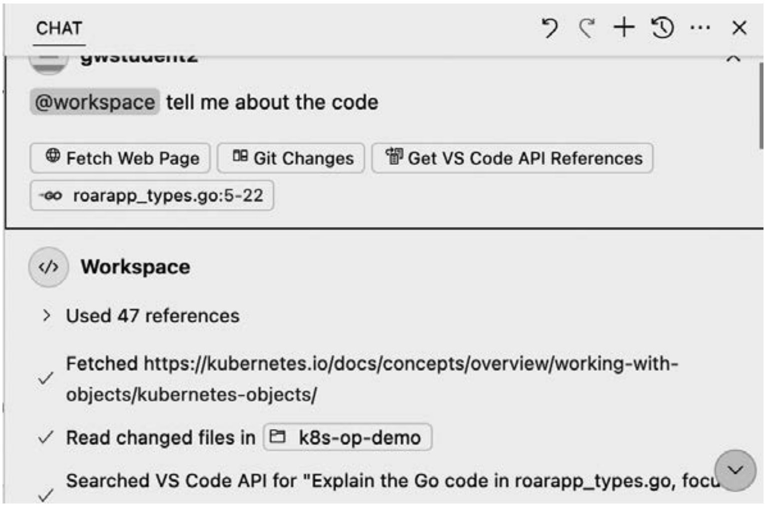


Рис. 3.14. Добавление дополнительного контекста

Чтобы найти конкретный элемент в списке, прокрутите его или введите текст в поле поиска. Список содержит файлы из вашего проекта. Кроме того, можно прокрутить список к концу, чтобы просмотреть файлы, открывавшиеся недавно.

Добавив нужные элементы контекста, введите промпт и отправьте его. При подготовке ответа Copilot проанализирует элементы. На рис. 3.15 показан пример вывода, полученного для промпта с несколькими добавленными видами контекста.



@workspace tell me about the code

Used 47 references

✓ Fetched https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

✓ Read changed files in k8s-op-demo

✓ Searched VS Code API for "Explain the Go code in roarapp_types.go, focu

Рис. 3.15. Выполнение промпта с разными типами контекста

Индексирование репозиториев

Напомню, что GitHub автоматически делает индексирование рабочих пространств, чтобы помочь разработчикам проводить быстрый и точный поиск нужных фрагментов кода в проекте. Результат индексирования можно использовать для поиска по более широкому диапазону контента в кодовой базе и получения более содержательных ответов.

Выбор участников чата

Элемент управления @ позволяет вызвать участника чата. Участники задают область применения промпта или запроса — например, все рабочее пространство, терминал или VS Code. В IDE доступно несколько встроенных участников чата — например, `@workspace` и `@terminal`.

Кроме того, дополнительные участники могут предоставляться GitHub Copilot Extensions после установки этих расширений. *Copilot Extensions* — нестандартные интеграции, добавляющие другие инструменты (такие как Docker) или API в чат-интерфейс вашей IDE. О Copilot Extensions мы поговорим в главе 10. На рис. 3.16 показан список встроенных участников и расширений, из которого можно выбрать нужный вариант.

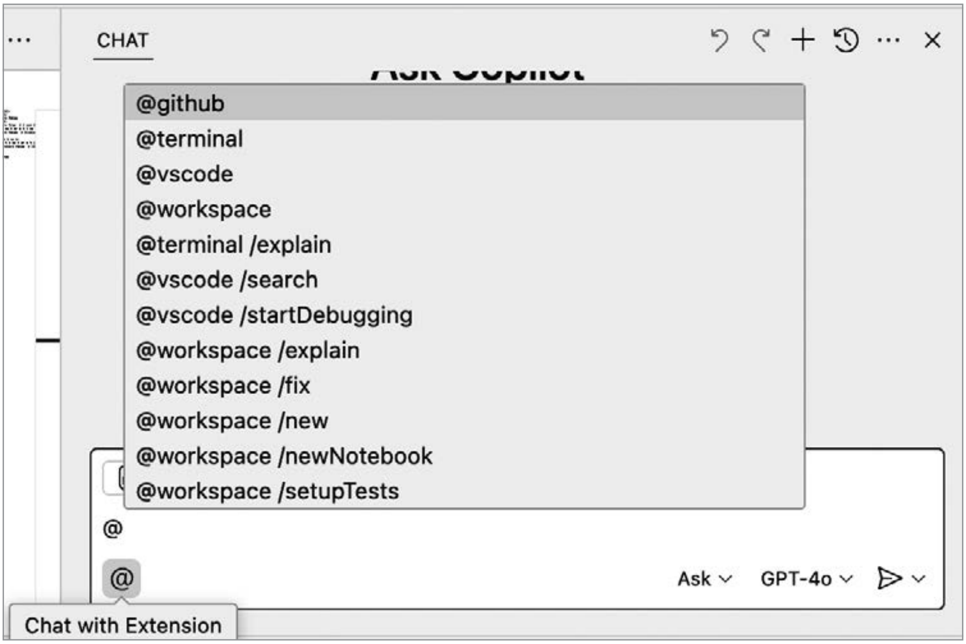


Рис. 3.16. Выбор участника или расширения

В нижнем правом углу диалогового окна Prompt находится кнопка Send. Нажав ее, вы отправляете промпт или запрос искусственному интеллекту, но сделать это можно несколькими способами. О них и поговорим далее (а к теме участников чата мы еще вернемся позднее в этой главе).

Способы отправки промпта

Кнопка со стрелкой в правой части области ввода промпта называется Send. Она позволяет выполнить обычную отправку, а также открывает доступ к нескольким другим вариантам отправки. Все команды Send, показанные на рис. 3.17, отправляют запрос или промпт ИИ-помощнику, но между ними есть некоторые ключевые различия. Разберемся, что делает каждая команда.

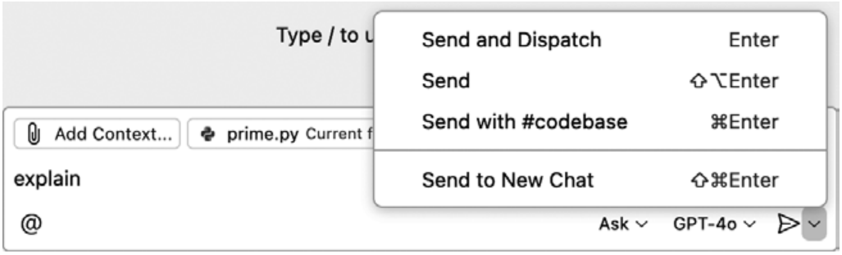


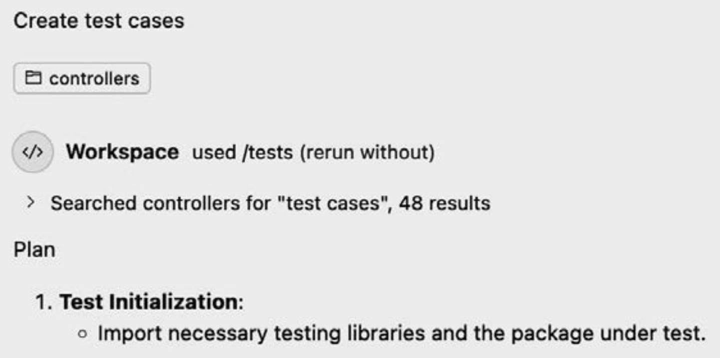
Рис. 3.17. Способы отправки

По умолчанию Copilot Chat пытается направить ваш промпт или запрос подходящему действию или инструменту (например, участнику чата), если это возможно. Это стандартное поведение доступно при выборе команды Send and Dispatch.

При перенаправлении действию или инструменту в верхней части вывода может появиться ссылка rerun without. На рис. 3.18 показано, что был введен промпт `Create test cases`. Но обратите внимание на то, что Copilot добавил участника `workspace` и быструю команду `/tests` (эти виды команд мы рассмотрим позднее в этой главе). Ссылка rerun without выполнит запрос без команды `/tests`.

Если автоматически выбранное действие или инструмент не подходит для вашего вопроса, то выберите ссылку, чтобы повторно отправить вопрос Copilot, не добавляя участника. Это поведение команды Send, если выбрать ее в раскрывающемся списке. На рис. 3.19 показано, как выглядит эта команда с тем же промптом `Create test cases`.

В третьей команде, Send with #codebase, `#codebase` — это чат-переменная. О таких переменных мы еще поговорим далее в главе. По сути, эта команда дает Copilot указание принять во внимание всю известную информацию обо всем вашем коде. Это может быть полезно, если в ответе должен учитываться другой контент из проекта, но вы не знаете, как задать отдельные элементы для контекста (или не хотите этого делать). Если вернуться к примеру с генерированием тестов, то, используя эту команду, вы поможете ИИ определить, есть ли уже тесты, и/или предложить варианты более комплексного тестирования в рамках проекта.



Create test cases

controllers

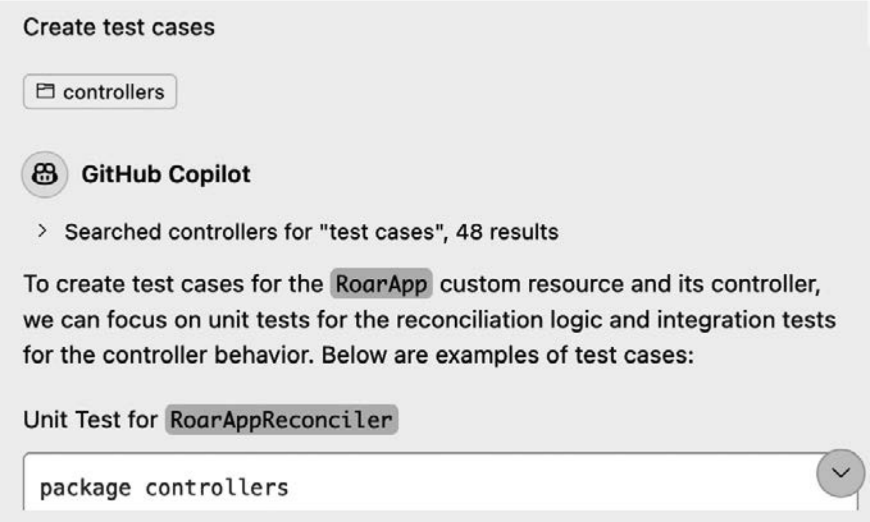
Workspace used /tests (rerun without)

Searched controllers for "test cases", 48 results

Plan

1. **Test Initialization**:
   - Import necessary testing libraries and the package under test.

Рис. 3.18. Пример промпта с автоматическими добавлениями, внесенными Copilot



Create test cases

controllers

GitHub Copilot

Searched controllers for "test cases", 48 results

To create test cases for the `RoarApp` custom resource and its controller, we can focus on unit tests for the reconciliation logic and integration tests for the controller behavior. Below are examples of test cases:

Unit Test for `RoarAppReconciler`

```
package controllers
```

Рис. 3.19. Пример отправки промпта с помощью команды Send

Последняя команда, Send to New Chat, понятна без пояснений. Только что введенный промпт откроется как первый промпт в новом чате. Этот вариант может быть полезен, если вы хотите отделить конкретный запрос или задачу от других обсуждений, чтобы не отвлекаться на них.

Выбор модели ИИ

В диалоговом окне Prompt есть элемент управления, позволяющий выбрать LLM, которая лучше подойдет для вашей конкретной задачи и должна использоваться в работе Copilot. В настоящее время существует множество моделей, которые можно выбрать в раскрывающемся списке (рис. 3.20). Со временем появятся новые модели, и в течение какого-то периода они будут помечены как предназначенные для предварительного ознакомления (preview). Кроме того, возможно, в рамках использования другой модели вам будет предложено разрешить доступ для всех клиентов (Enable access for all clients).

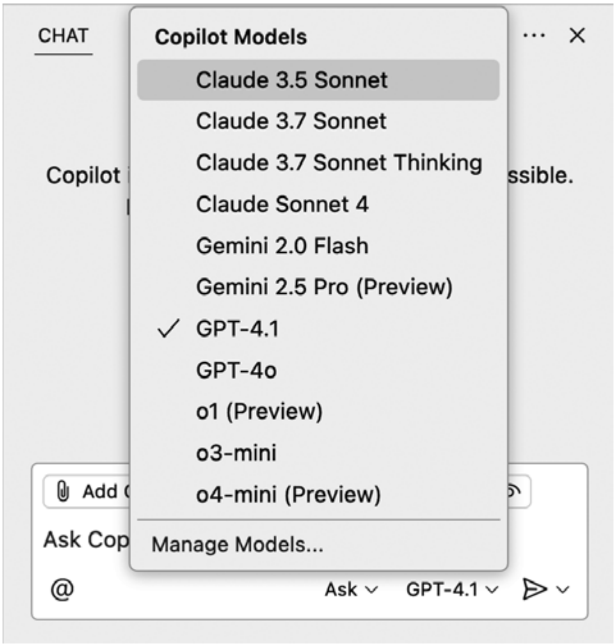


Рис. 3.20. Выбор модели

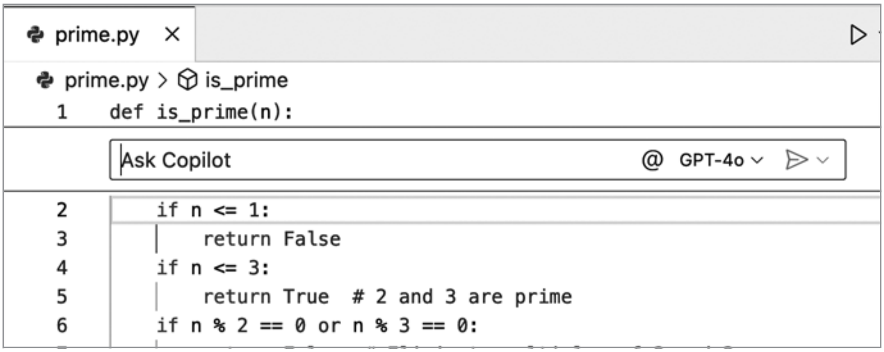
Управление моделями

Возможно, вы заметили ссылку Manage Models в нижней части списка моделей на рис. 3.20. Она позволяет ввести данные, которые могут понадобиться для обращения к другим моделям, например идентификатор модели или ключи API.

Итак, вы увидели, как работает главное диалоговое окно чата. Теперь рассмотрим способ использования чата напрямую в редакторе кода.

Использование встроенного чат-интерфейса в редакторе

Интеграция Copilot предоставляет упрощенный интерфейс быстрого доступа к чату прямо в редакторе. Интерфейс можно вызвать в любом месте кода, нажав Meta+I (рис. 3.21).



```
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True  # 2 and 3 are prime
    if n % 2 == 0 or n % 3 == 0:
```

Рис. 3.21. Встроенный чат-интерфейс

Это диалоговое окно представляет собой уменьшенную версию главного окна чата. Оно было разработано для использования прямо в редакторе кода так, чтобы не занимать слишком много места на экране. Помимо возможности добавления контекста, оно содержит те же элементы управления, и вы можете вводить те же промпты или запросы, что и в главном окне чата. Например, можно выделить код и задать тот же вопрос, который задавался ранее в этой главе:

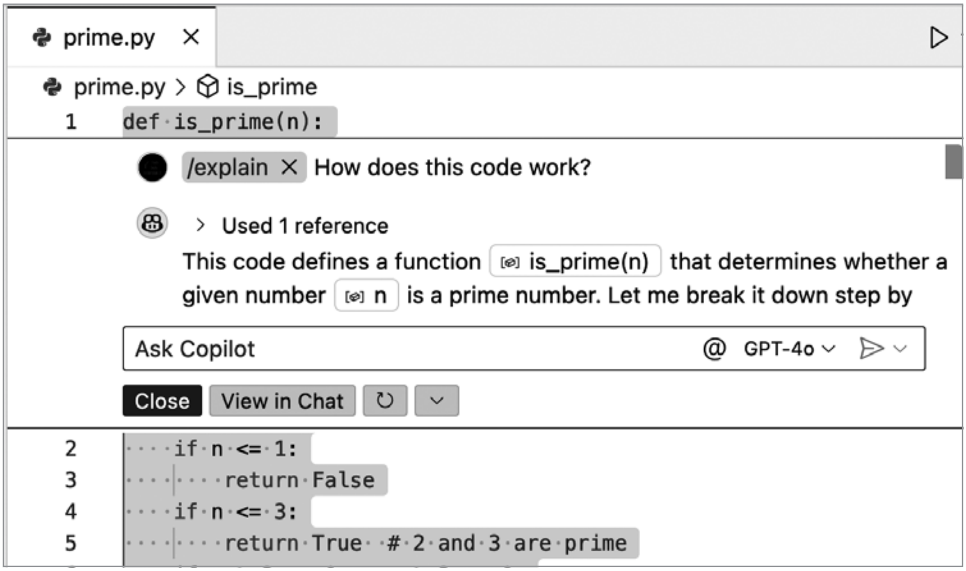
```
Как работает этот код?
```

Copilot покажет такую часть ответа, которая поместится в ограниченном окне (рис. 3.22). Чтобы увидеть скрытую часть вывода, прокрутите окно или разверните его. Возможно, что из-за нехватки места на экране вы все равно не увидите полный ответ. Чтобы увидеть его в главном интерфейсе чата, нажмите кнопку View in Chat.

Повторное генерирование ответа

Возможно, вы заметили круглую стрелку рядом с кнопкой View in Chat. Этот символ означает, что вы можете попросить Copilot генерировать ответ повторно, чтобы посмотреть, не предложит ли он другой, более удачный вариант. Эта возможность доступна не только в этом окне.

Нажатие кнопки View in Chat приводит к выводу полного текста ответа в главном окне чата (рис. 3.23). Вы сможете прокрутить ответ, вводить уточняющие промпты, выделять код и копировать/вставлять примеры кода.



```
def is_prime(n):
```

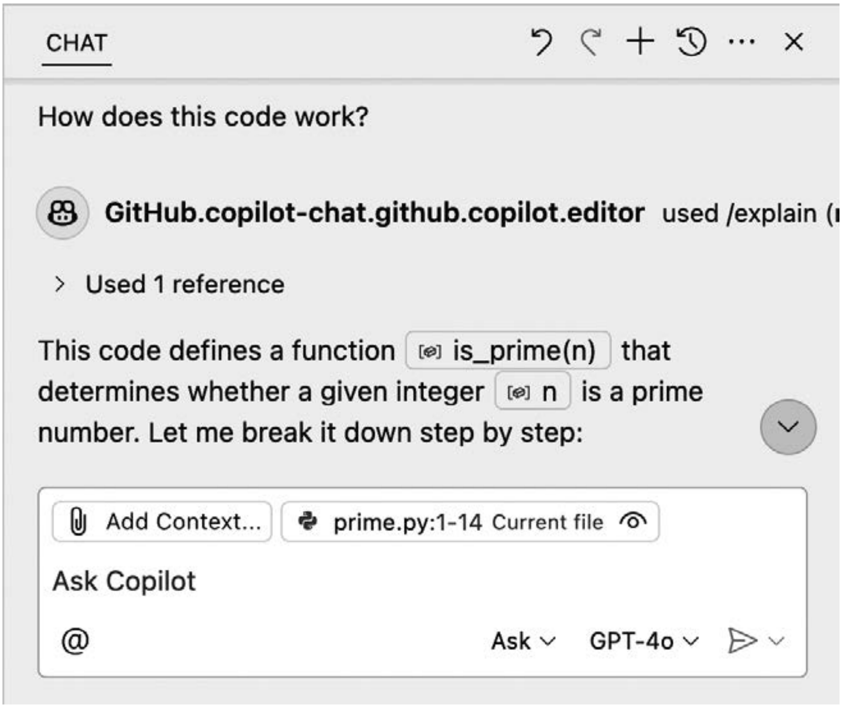
/explain How does this code work?

Used 1 reference

This code defines a function is_prime(n) that determines whether a given number n is a prime number. Let me break it down step by

```
    if n <= 1:
        return False
    if n <= 3:
        return True  # 2 and 3 are prime
```

Рис. 3.22. Введение промпта в интерактивном чате



How does this code work?

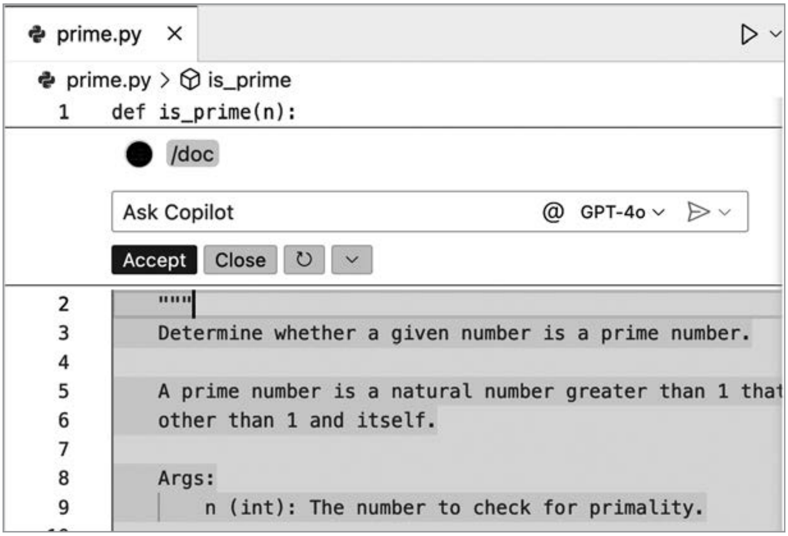
GitHub.copilot-chat.github.copilot.editor used /explain (

Used 1 reference

This code defines a function is_prime(n) that determines whether a given integer n is a prime number. Let me break it down step by step:

Рис. 3.23. Вывод, полученный с помощью команды View in Chat

Другие операции, запущенные из встроенного диалогового окна, могут обновлять код непосредственно в редакторе. Например, если вы попросите документировать выделенный код, то Copilot сгенерирует для него документацию (рис. 3.24).



```
def is_prime(n):
    """
    Determine whether a given number is a prime number.

    A prime number is a natural number greater than 1 that
    other than 1 and itself.

    Args:
        n (int): The number to check for primality.
```

Рис. 3.24. Предлагаемые изменения в документации

Таким образом, если в промпте есть указание для Copilot изменить какой-либо элемент реального кода в редакторе, то результаты будут применены напрямую, чтобы вы могли проверить их и решить, подтвердить их или нет. Если промпт запрашивает ответ, который представляет альтернативу коду в IDE (например, если вы попросили перевести код на Go или оптимизировать его), то вывод может появиться либо в интерактивном окне чата (если ответ достаточно короткий), либо в главной области вывода в чате.

Главная панель чата и встроенный чат в редакторе подходят для большинства случаев использования, но есть еще один вариант для быстрого взаимодействия с чатом — *быстрый чат* (Quick Chat).

Использование интерфейса быстрого чата

Иногда вам может понадобиться быстро задать Copilot вопрос или отправить промпт, который не относится к коду в вашем текущем редакторе и не требует перехода на главную панель чата. Для таких случаев в Copilot есть функция Quick Chat, которая вызывает простое диалоговое окно чата в верхней части IDE (рис. 3.25).

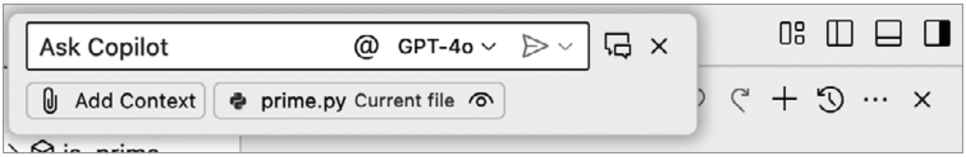


Рис. 3.25. Быстрый чат

Само диалоговое окно похоже на то, что используется во встроенном редакторе, но не размещается в нем. Благодаря Quick Chat вы можете взаимодействовать с Copilot через промпт, а затем вернуться к предыдущим действиям. Чтобы вызвать быстрый чат, нажмите Meta+I или щелкните на значке Copilot в верхней части интерфейса VS Code. При щелчке на этом значке (рис. 3.26) открывается меню с командами для открытия главной панели чата, диалогового окна Quick Chat и встроенного чата Editor Inline Chat.

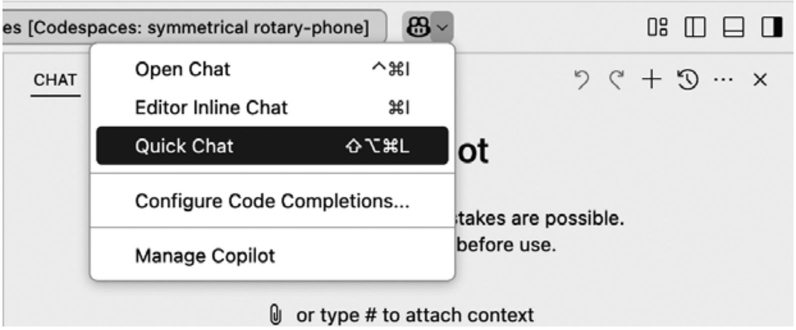


Рис. 3.26. Команды открытия чата в меню Copilot

Чтобы сделать взаимодействие с Copilot более эффективным, для типичных запросов к Copilot можно использовать быстрые команды. Они описаны в следующем разделе.

Быстрые команды для взаимодействия с чатом

GitHub Copilot Chat поддерживает несколько команд для основных типов взаимодействия. Эти быстрые команды называются *слеш-командами* (slash commands)[1], поскольку начинаются с символа / (слеш). Они работают как из главного окна чата, так и из встроенного интерфейса чата (который можно вызвать, нажав со-

---

[1] Подобные команды используются не только в Copilot, но и в чат-ботах Telegram и на других платформах.

четание клавиш Meta+I). Не все команды доступны в каждой IDE, но в табл. 3.1 приведен полный набор этих команд, доступных на момент написания книги.

Таблица 3.1. Слеш-команды для работы с Copilot

Команда	Действие
`/clear`	Очистить текущий диалог с Copilot
`/doc`	Добавить комментарии для выделенного или указанного кода
`/edit`	Редактировать выделенный код в активном редакторе
`/explain`	Сгенерировать объяснения по работе кода
`/fix`	Предложить решение проблемы
`/fixTestFailure`	Предложить решение для проваленного теста
`/generate`	Сгенерировать код для ответа на конкретный вопрос
`/help`	Выдать справку по работе с Copilot
`/new`	Создать каркасный код для нового рабочего пространства
`/newNotebook`	Создать новый блокнот Jupyter
`/search`	Найти контент в VS Code
`/startDebugging`	Сгенерировать конфигурацию запуска и начать отладку в VS Code
`/tests`	Создать юнит-тесты для кода

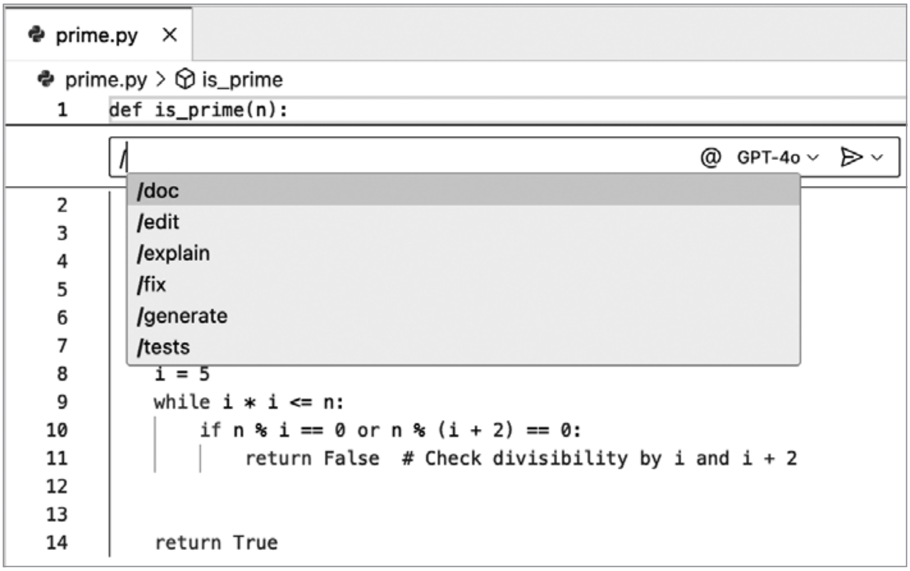
Ограниченная функциональность

В разных IDE может быть доступна только часть этих команд или те, которые не указаны в этой таблице. Набор реализованных и поддерживаемых слеш-команд может часто изменяться, поэтому всегда обращайтесь к документации Copilot по вашей IDE, чтобы узнать, какие команды доступны в этой среде.

Использовать эти команды несложно. Введите символ `/` в одном из чат-интерфейсов; Copilot выведет список доступных команд (рис. 3.27). Учтите, что не все они доступны во встроенном чат-интерфейсе.

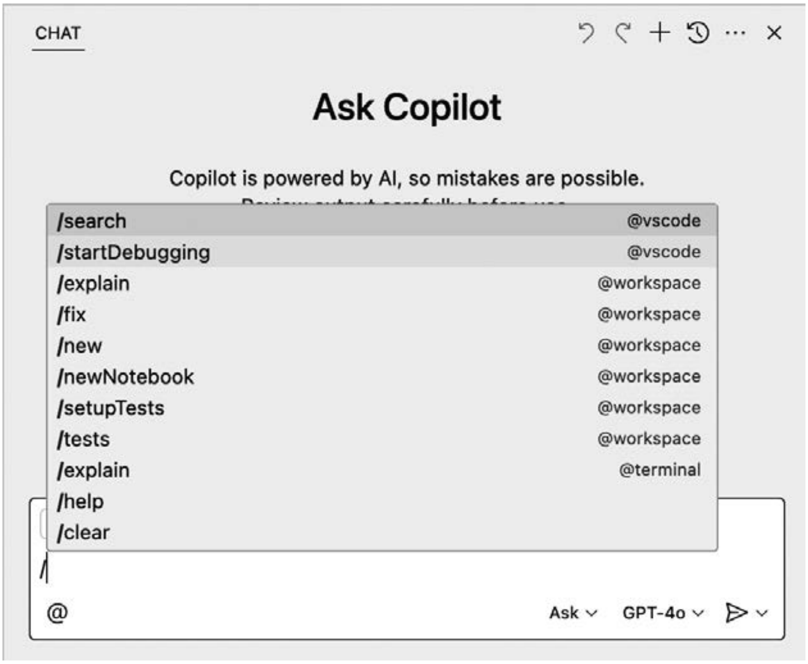
Тот же способ можно использовать на главной панели чата (рис. 3.28).

В этом разделе мы очень кратко обсудили слеш-команды, но они могут быть весьма полезными. О некоторых из них мы подробнее поговорим в следующих главах.



```
def is_prime(n):
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False  # Check divisibility by i and i + 2


    return True
```

Рис. 3.27. Вывод списка слеш-команд в интерактивном чате



Ask Copilot

Copilot is powered by AI, so mistakes are possible.

Рис. 3.28. Вывод списка слеш-команд на главной панели чата

Функция объяснения кода лучше всего работает в IDE, в которых установлены соответствующие расширения для используемого языка. На момент написания книги для VS Code были доступны расширения для следующих языков:

- TypeScript/JavaScript;
- Python;
- Java;
- C#;
- C++;
- Go;
- Ruby.

В строках со слеш-командами (см. рис. 3.28) находятся элементы управления, начинающиеся с символа `@`. Как упоминалось ранее, они называются *участниками* и помогают определить *область применения* и *предметную область*, в которых можно использовать быстрые команды. Этим элементам управления посвящен следующий раздел.

Участники чата

В GitHub Copilot *участники чата* (chat participants) фактически представляют собой указатели на определенные предметные области. Добавляя их в промпт в чат-интерфейсе, вы фактически даете ИИ указание использовать для генерации релевантных ответов конкретную предметную область и знания находящегося в ней участника.

Чтобы добавить ссылку на одного из участников чата, введите символ `@` и имя участника, который должен использоваться в промпте. На момент написания книги по данным документации GitHub были доступны следующие участники (https://oreil.ly/v0714).

`@workspace`

Содержит знания о коде в рабочем пространстве. Используйте в ситуациях, когда хотите, чтобы Copilot учитывал структуру вашего проекта, способы взаимодействия разных частей кода или паттерны проектирования в вашем проекте.

`@vscode`

Содержит знания о командах и функциональности VS Code. Используйте, если вам нужна помощь с VS Code.

`@terminal`

Содержит знания об оболочке терминала VS Code и ее содержимом. Используйте, если вам нужна помощь с созданием или отладкой команд в терминале.

`@vision`

Позволяет прикрепить файл с изображением, чтобы помочь Copilot понять вопрос. (На стадии предварительного ознакомления.)

`@github`

Позволяет запрашивать информацию по проблемам, pull-запросам и другим элементам вашего репозитория. (Доступен не во всех интерфейсах.)

`@project`

(Только для JetBrains IDE.) Как и `@workspace`, содержит знания о коде в вашем проекте.

Установка других участников чата

Помимо использования встроенных участников чата, предоставляемых GitHub, в Copilot можно устанавливать расширения Copilot для работы с дополнительными нестандартными участниками чата. О том, как это делается, а также о расширении возможностей Copilot мы поговорим в главе 10.

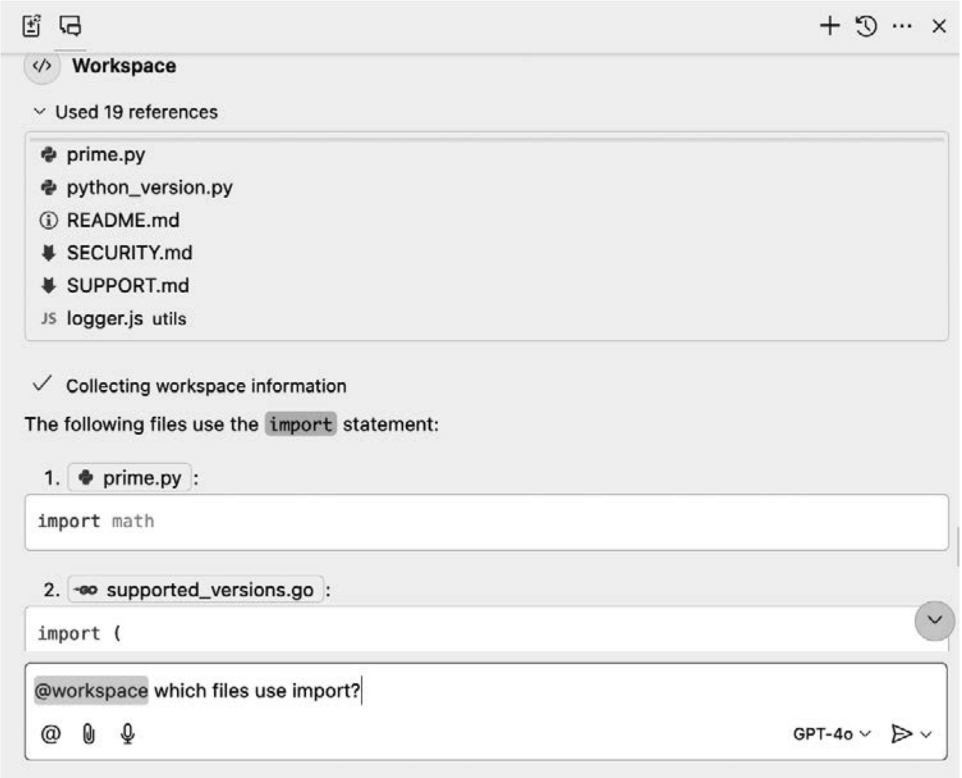
А теперь перейдем к наиболее распространенным участникам чата.

@workspace

Этот участник полезен, если вы хотите, чтобы Copilot явно учитывал весь код, находящийся в рабочем пространстве, как потенциальный источник для формирования ответа в чате. Это не означает, что в процессе создания ответа будет использоваться весь код. Расширение Copilot учитывает все файлы, являющиеся частью рабочего пространства, чтобы определить, какие файлы (или их фрагменты) могут быть полезными для контекста. Автоматическое индексирование, о котором я упоминал ранее, упрощает этот процесс.

Допустим, вы через `@workspace` спрашиваете у Copilot, в каких файлах используется инструкция `import`. Сначала Copilot соберет информацию о рабочем пространстве, чтобы определить релевантный контент. Сделав это, Copilot выведет список ссылок, которые, по его мнению, стоит проверить, учитывая введенный промпт. Этот список будет показан в начале ответа. Как я говорил ранее, для просмотра списка выбранных ссылок можно развернуть строку Used … references. Мы используем участника `@workspace`, так что на этот раз совпадений будет больше. Более обширный список ссылок, полученных с помощью промпта, в котором используется `@workspace`, показан на рис. 3.29.

Когда в промптах используется участник чата `@workspace`, мы получаем ответ, который охватывает рабочее пространство и содержит список из нескольких файлов.



Workspace

Used 19 references

- prime.py
- python_version.py
- README.md
- SECURITY.md
- SUPPORT.md
- logger.js utils

✓ Collecting workspace information

The following files use the `import` statement:

1. prime.py :

```
import math
```

2. supported_versions.go :

```
import (
```

@workspace which files use import?

Рис. 3.29. Запрос файлов, в которых используется инструкция import, с помощью участника @workspace

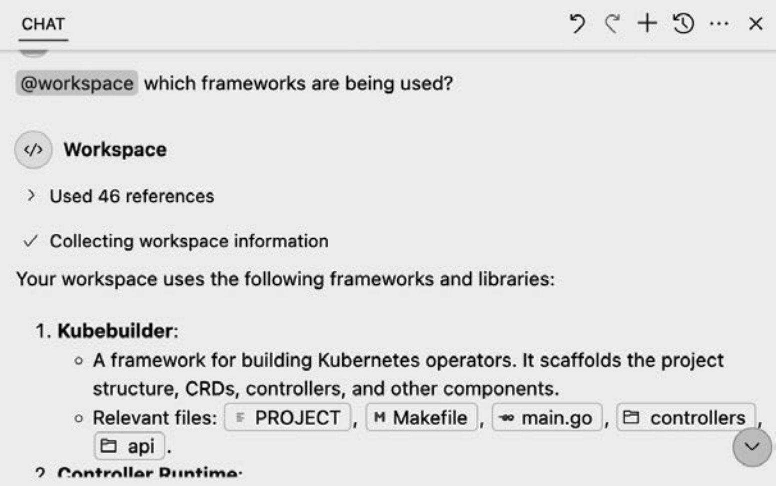
Изменения при работе с @workspace

Функциональность @workspace использует сохраненный контент вашей кодовой базы, поэтому перед выполнением промпта важно сохранить любые изменения или новый контент, который должен учитываться участником чата.

С помощью `@workspace` можно задавать более общие вопросы о блоке кода. Это может быть полезно для поиска информации о контенте, который вам не знаком. Например, можно задавать такие общие вопросы.

- «Какая бэкенд-система используется в проекте?»
- «Как выполнить этот код?»
- «Где используется ...?»
- «Какой сервис использовать для выполнения ...?»
- «Какие фреймворки здесь используются?»

На рис. 3.30 показан пример использования последнего запроса для получения информации.



@workspace which frameworks are being used?

Used 46 references

✓ Collecting workspace information

Your workspace uses the following frameworks and libraries:

1. **Kubebuilder**:
   - A framework for building Kubernetes operators. It scaffolds the project structure, CRDs, controllers, and other components.
   - Relevant files: PROJECT, Makefile, main.go, controllers, api.
2. **Controller Runtime**:

Рис. 3.30. Пример использования участника @workspace

Сочетание клавиш для добавления @workspace

Если использовать Meta+Enter (вместо простого Enter) при отправке запроса в чате, то участник @workspace добавится в промпт автоматически.

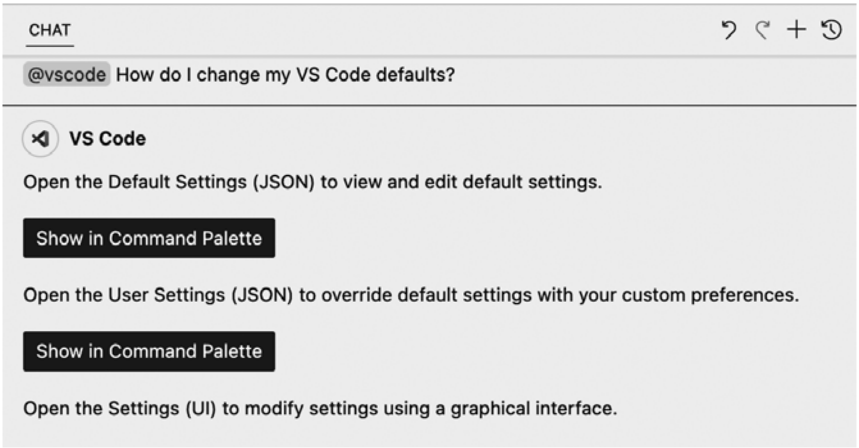
Итак, `@workspace` предоставляет контекст, связанный с блоком кода. В свою очередь, с помощью `@vscode` можно составлять вопросы или промпты, связанные с использованием VS Code.

@vscode

Этот участник чата (поддерживается только в интегрированной среде VS Code) выполняет функции эксперта по командам и настройкам VS Code. Используя его в чат-интерфейсе Copilot, можно задавать вопросы о VS Code, вводя их на естественном языке. В текущей реализации этот участник чата имеет доступ к индексу всех настроек и команд приложения. Применяя `@vscode`, можно задавать в интерфейсе GitHub Copilot Chat такие вопросы.

- «Как открыть проект в VS Code?»
- «Как устанавливать расширения в VS Code?»
- «Как создать новый файл в VS Code?»
- «Как сменить тему в VS Code?»

На рис. 3.31 показан пример использования `@vscode` для получения информации о том, как изменить стандартные настройки VS Code.



@vscode How do I change my VS Code defaults?

VS Code

Open the Default Settings (JSON) to view and edit default settings.

Open the User Settings (JSON) to override default settings with your custom preferences.

Open the Settings (UI) to modify settings using a graphical interface.

Рис. 3.31. Использование @vscode в чате

Кроме того, на этом снимке экрана видно, что Copilot добавляет ссылку, по которой можно перейти к другому интерфейсу (палитре команд) для ускоренного вызова функций. Нажатие кнопки в выводе чата открывает соответствующие настройки.

Приведу еще один пример использования участника `@vscode`. Представьте, что вы хотите получить информацию о том, как выполнять отладку в вашей IDE. Для этого введите следующий запрос:

```
@vscode как выполнить отладку
```

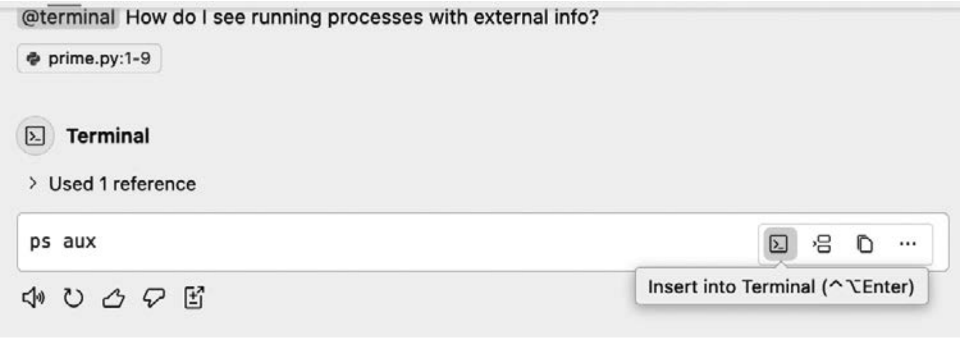
Последний участник чата, которого мы рассмотрим в этой главе, полезен для взаимодействия с терминалом.

@terminal

Этот участник позволяет задать Copilot вопросы об интегрированной оболочке терминала, ее буфере и текущем выделении в терминале. Это поможет понять, какие действия были выполнены в терминале, выявить любые проблемы и получить доступ к нужным командам. Вот примеры вопросов/промптов, которые можно ввести, используя `@terminal`.

- «Что сделала последняя команда?»
- «Объясни команду».
- «Как мне ...?»

На рис. 3.32 показан простой пример использования `@terminal`: задан вопрос о том, как выполнить действие с помощью команды в терминале. Если навести указатель мыши на вывод, то справа появляется всплывающий элемент управления, позволяющий вставить ответ прямо в терминал.



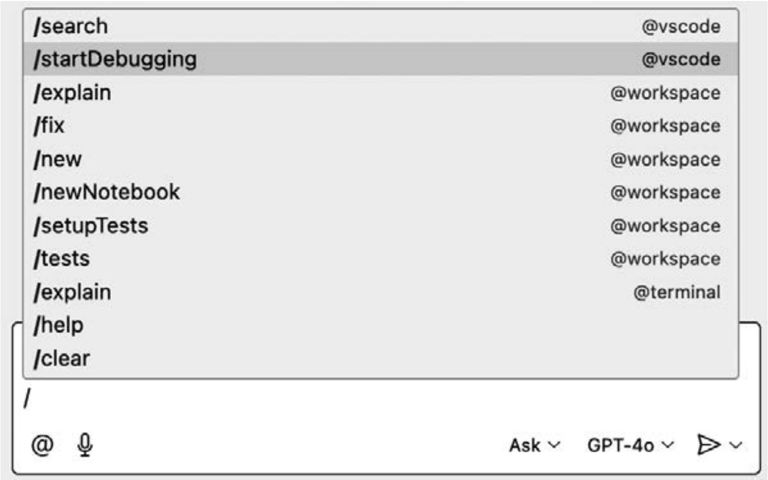
@terminal How do I see running processes with external info?

```
ps aux
```

Рис. 3.32. Использование участника чата @terminal

Кроме того, с помощью `@terminal` можно задать Copilot вопросы о стандартных последовательностях операций, которые выполняются в терминале. Например, вы можете спросить, как сохранить ваш код в репозитории при использовании Git в командной строке.

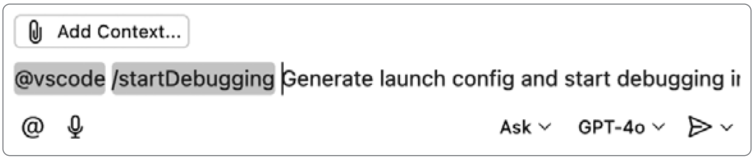
Многие слеш-команды по умолчанию привязаны к одному из участников чата как к предметной области для выполнения команд. В примере с отладкой VS Code (см. выше) с `@vscode` связана слеш-команда `/startDebugging` (рис. 3.33).



/search	@vscode
/startDebugging	@vscode
/explain	@workspace
/fix	@workspace
/new	@workspace
/newNotebook	@workspace
/setupTests	@workspace
/tests	@workspace
/explain	@terminal
/help	
/clear	

Рис. 3.33. Участники и связанные с ними слеш-команды

Таким образом, если ввести ее в поле запроса, то к ней автоматически добавится префикс `@vscode` (рис. 3.34). Это означает, что команда `/startDebugging` в настоящее время актуальна только применительно к VS Code.



`@vscode /startDebugging Generate launch config and start debugging i`

Рис. 3.34. Автоматическая вставка @vscode при выборе команды /startDebugging

В табл. 3.2 показано, с каким участником связана та или иная слеш-команда, а также для чего нужна каждая из них.

Таблица 3.2. Слеш-команды и контекст в Copilot

Команда	Участник	Действие
`/explain`	`@terminal`	Объяснить происходящее в терминале
`/search`	`@vscode`	Сгенерировать параметры запроса для поля Search
`/startDebugging`	`@vscode`	Сгенерировать конфигурацию запуска и начать отладку в VS Code
`/explain`	`@workspace`	Объяснить, как работает активный код
`/fix`	`@workspace`	Предложить исправление для ошибок в вашем коде
`/fixTestFailure`	`@workspace`	Предложить исправление для проваленного теста
`/new`	`@workspace`	Создать каркас кода для нового рабочего пространства или нового файла
`/newNotebook`	`@workspace`	Создать новый блокнот Jupyter с описанием на естественном языке
`/setupTests`	`@workspace`	Сгенерировать тесты в проекте
`/tests`	`@workspace`	Сгенерировать модульные тесты для кода
`/help`		Выдать справку по использованию GitHub Copilot
`/clear`		Создать новый сеанс чата

С помощью участников чата можно задавать более обширный контекст. Но иногда может потребоваться уточнить контекст. В этом случае можно использовать специальные идентификаторы для указания файла, контента в редакторе, выделения и т. д. — *чат-переменные*.

Чат-переменные

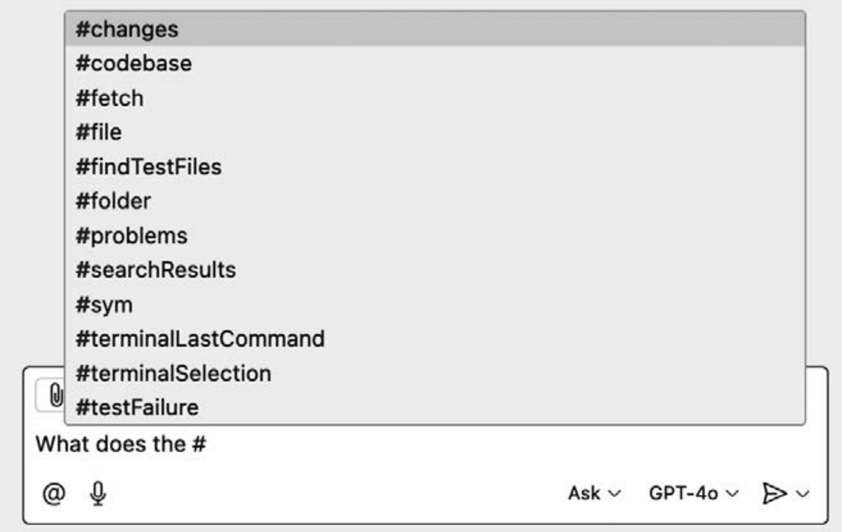
Чат-переменные позволяют уточнить контекст непосредственно в тексте промпта. Они могут ссылаться на элементы в редакторе, конкретный файл, репозиторий Git, команды терминала и многое другое. Чтобы сослаться на чат-переменную в промпте через чат-интерфейс, используйте префикс `#`. Например, `#terminalSelection` добавляет выделенный фрагмент из активного терминала.

В табл. 3.3 перечислены чат-переменные, доступные на момент написания книги (актуальную информацию см. в документации VS Code (https://code.visualstudio.com/docs/copilot/reference/copilot-vscode-features?t#_chat-tools)). В столбце «Контент» указано, что добавляется в проект в качестве контекста для вашего промпта.

Таблица 3.3. Чат-переменные

Чат-переменная	Контент
`#changes`	Список изменений в системе контроля версий
`#codebase`	Релевантный контент, найденный в вашем рабочем пространстве
`#fetch`	Контент с веб-страницы, заданной URL
`#`*`<имя_файла/имя_папки>`*	Файл или папка, которую вы хотите добавить к контексту
`#problems`	Проблемы, обнаруженные в рабочем пространстве (см. панель Problems); может использоваться как контекст для отладки или исправления кода
`#searchResults`	Результаты поискового запроса, выполненного в вашем рабочем пространстве
`#selection`	Текущий выделенный текст в редакторе — добавляется в формате *`<имя_файла:строки>`*
`#`*`<символ>`*	Вывод списка символов (соответствующих шаблону, введенному после #), из которого вы выбираете нужные
`#terminalLastCommand`	Последняя выполненная команда в активном терминале
`#terminalSelection`	Выделение в активном терминале
`#testFailure`	Добавление информации о сбоях тестов, упрощающей диагностику и исправление тестов
`#usages`	Объединение команд Find All References, Find Implementation и Go to Definition
`#vscodeAPI`	Добавление VS Code API для ответов на вопросы о разработке расширений VS Code

Чтобы получить доступ к чат-переменным, просто введите в промпте символ #. Copilot выводит список доступных переменных (рис. 3.35).



What does the #

Рис. 3.35. Чат-переменные

После этого можно выбрать из полученного списка нужный элемент (рис. 3.36).

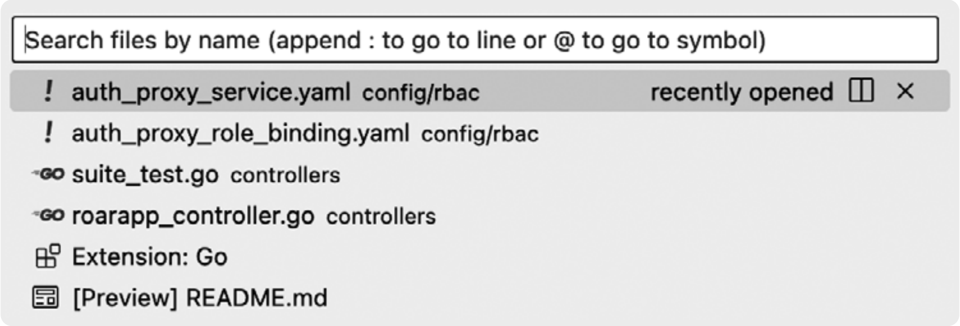
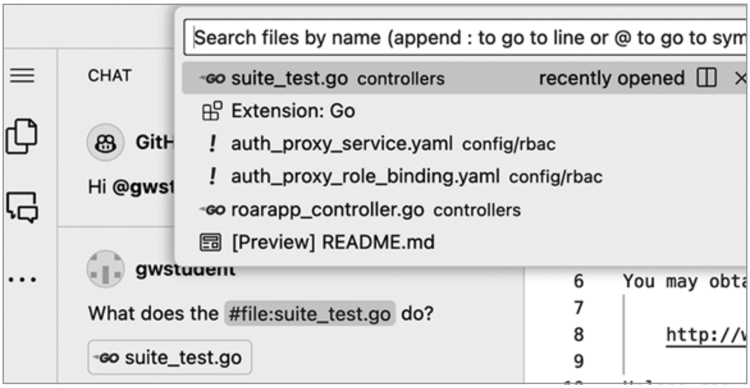


Рис. 3.36. Выбор файла из заполненного списка

Кроме того, в промптах можно использовать комбинации из участников, слеш-команд и чат-переменных. На рис. 3.38 показан пример обращения к Copilot с просьбой объяснить выделенный код в файле. Обратите внимание, что `@workspace` подставляется автоматически, поэтому промпт теперь выглядит так: `@workspace /explain #selection`.

ИСПОЛЬЗОВАНИЕ ПРАВИЛЬНЫХ ПУТЕЙ

При использовании переменной `#file` и выборе файла из полученного списка достаточно выбрать имя файла (рис. 3.37).

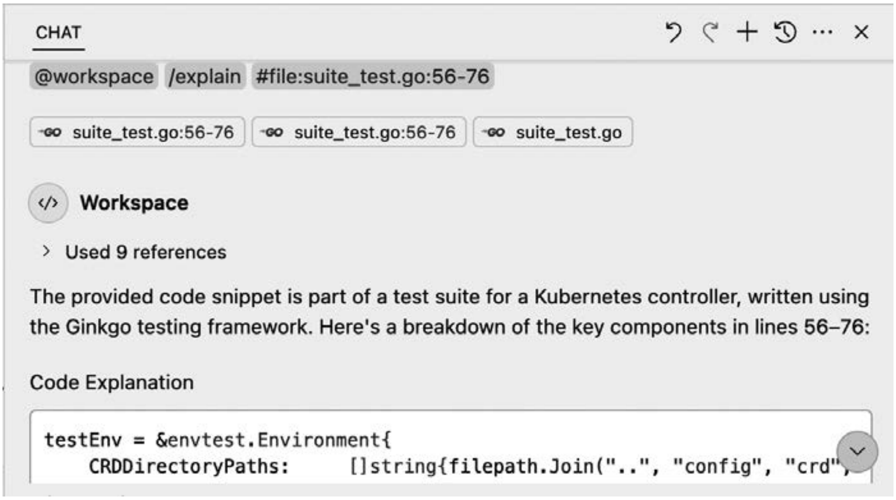


What does the #file:suite_test.go do?

Рис. 3.37. Выбор имени файла из списка

Но если вы вводите имя файла как часть промпта, то не забудьте добавить относительный путь от корня проекта:

```
Что делает #file:controllers/suite_test.go?
```



@workspace /explain #file:suite_test.go:56-76

The provided code snippet is part of a test suite for a Kubernetes controller, written using the Ginkgo testing framework. Here's a breakdown of the key components in lines 56–76:

Code Explanation

```
testEnv = &envtest.Environment{
    CRDDirectoryPaths:     []string{filepath.Join("..", "config", "crd"
```

Рис. 3.38. Определение контекста с помощью участника чата, слеш-команды и чат-переменной

Если вы посмотрите на используемые ссылки, то увидите, что Copilot в итоге сфокусировался на выделенном фрагменте и другой части файла, которая потребовалась ему для формирования ответа. Затем Copilot предоставил объяснение для более подходящего контекста (выделенный текст), как было указано в запросе.

Убедиться в том, что Copilot принимает во внимание нужный вам контекст, иногда достаточно сложно. Главное — внимательно изучить ответ и выяснить:

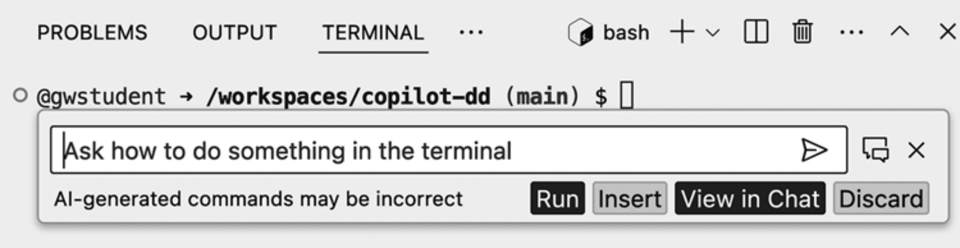
- использовал ли Copilot ссылки, которые, по вашему мнению, должен был использовать;
- является ли ответ *общей* рекомендацией, или он характерен только для структуры либо контента вашего рабочего пространства.

Ответы на эти вопросы помогут быстро определить, учитывает ли Copilot предложенный вами контент, или вам стоит использовать один из способов, описанных выше, чтобы задать более точную настройку.

Существует еще один чат-интерфейс, который тоже можно использовать при взаимодействии с Copilot. Он предназначен для операций, выполняемых в терминале.

Чат в терминале

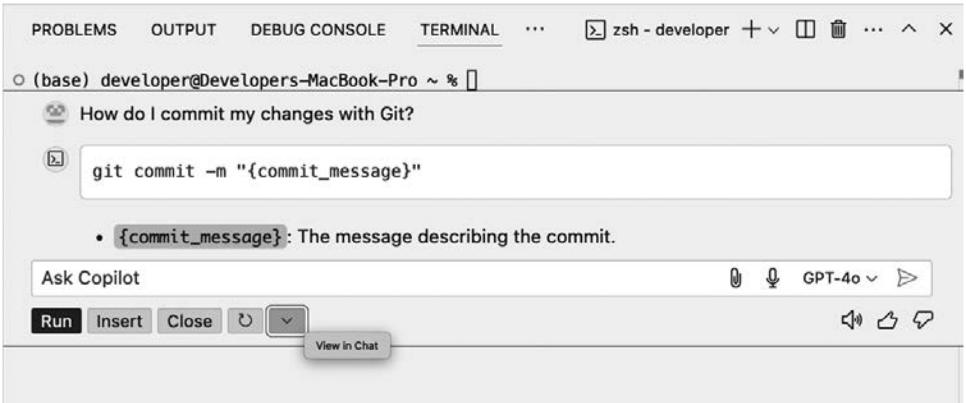
Copilot предоставляет специализированную функциональность чата, предназначенную для работы с терминалом. Когда вы работаете в терминале (например, в VS Code), при нажатии Meta+I открывается диалоговое окно для введения промптов, передаваемых терминалу (рис. 3.39).



```
○ @gwstudent → /workspaces/copilot-dd (main) $
```

Рис. 3.39. Диалоговое окно запроса в терминале

Например, вы можете ввести: «Как сохранить изменения в Git?», а Copilot выдаст команды, которые необходимо выполнить в терминале. Кроме того, вы можете запустить их, просмотреть вывод в чате, а также выполнить другие действия (рис. 3.40).



```
(base) developer@Developers-MacBook-Pro ~ %
```

How do I commit my changes with Git?

```
git commit -m "{commit_message}"
```

- `{commit_message}`: The message describing the commit.

Рис. 3.40. Просмотр чата в терминале

Создание пользовательских инструкций по генерированию кода

В главе 2 я упоминал о том, что вы можете создавать собственные инструкции Copilot по ревью кода. Помимо них, можно создавать пользовательские инструкции для управления генерированием ответов. Однако следует учесть, что такие инструкции не применяются к встроенным предложениям кода в редакторе.

Как и в случае специализированных инструкций по выполнению ревью кода, создать пользовательскую инструкцию по генерированию кода можно двумя способами.

Во-первых, можно отредактировать файл `settings.json` и добавить собственные инструкции в раздел `github.copilot.chat.codeGeneration.instructions`. Вот пример правила, добавленного в файл:

```
"github.copilot.chat.codeGeneration.instructions": [
    {
        "text": "Всегда добавлять в сгенерированный код комментарии, описывающие
                 его назначение."
    },
    {
        "text": "Использовать подчеркивания в именах приватных полей в TypeScript."
     }
]
```

Добавив правило, можно попросить Copilot выполнить ревью того же кода снова. На этот раз в выводе будет помечено условие, которое вы добавили в файл настроек (см. рис. 2.24).

Во-вторых, можно создать в репозитории или проекте Markdown-файл с конкретным путем и именем `.github/copilot-instructions.md`.

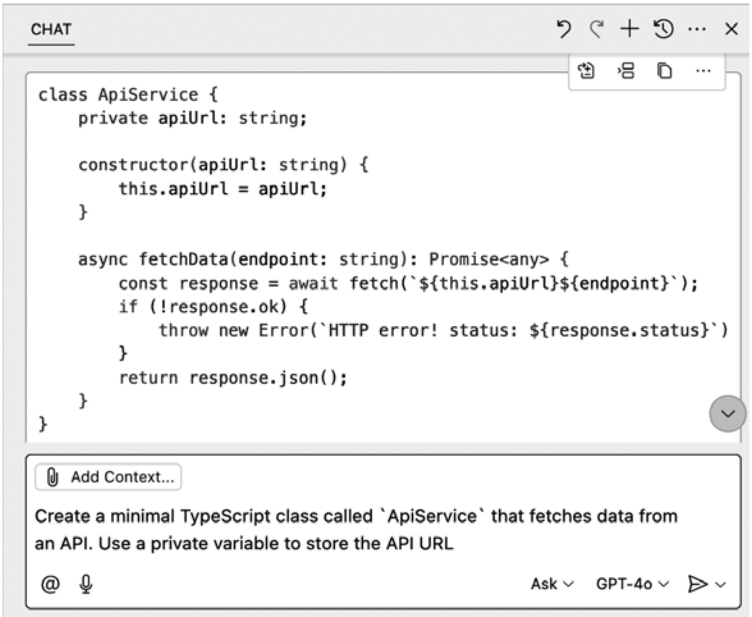
Вот пример такого файла с теми же правилами:

```
# Пользовательские инструкции для генерирования кода

- Всегда добавлять в сгенерированный код комментарии, описывающие его назначение.
- Использовать подчеркивания в именах приватных полей в TypeScript.
```

Если вы предпочитаете использовать другой путь или имя для Markdown-файла, то создайте этот файл и добавьте ссылку на него в файл `settings.json` с ключом `file:`. Например, если вы присвоили файлу имя `.docs/code-style.md`, то сможете сообщить Copilot о нем, добавив в файл настроек такую запись:

```
"github.copilot.chat.reviewSelection.instructions": [
   {
      "file": ".docs/code-style.md"
   }
]
```

Рассмотрим пример, в котором нет специализированных инструкций по генерированию кода. Допустим, вы просите Copilot создать минимальный класс TypeScript с приватной переменной (рис. 3.41).

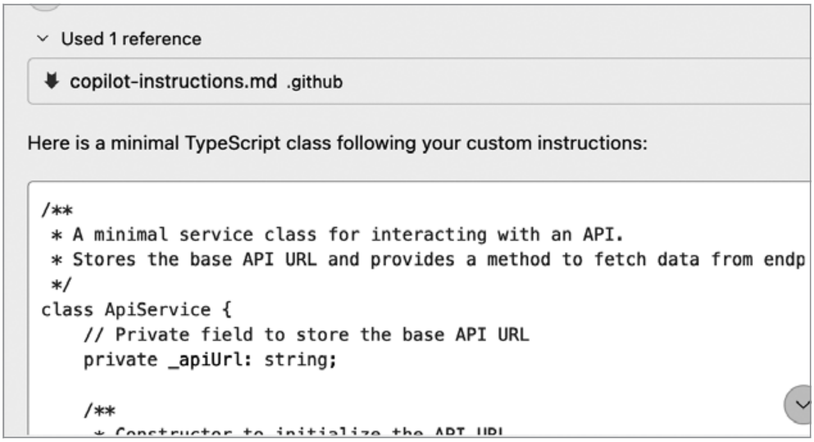


```
class ApiService {
    private apiUrl: string;

    constructor(apiUrl: string) {
        this.apiUrl = apiUrl;
    }

    async fetchData(endpoint: string): Promise<any> {
        const response = await fetch(`${this.apiUrl}${endpoint}`);
        if (!response.ok) {
            throw new Error(`HTTP error! status: ${response.status}`)
        }
        return response.json();
    }
}
```

Create a minimal TypeScript class called `ApiService` that fetches data from an API. Use a private variable to store the API URL

Рис. 3.41. Минимальный класс TypeScript без специализированных инструкций по генерированию кода

Допустим, вы добавили собственные инструкции, используя один из описанных выше методов. В таком случае при выполнении того же промпта вы получите результат, показанный на рис. 3.42.



Used 1 reference

copilot-instructions.md .github

Here is a minimal TypeScript class following your custom instructions:

```
/**
 * A minimal service class for interacting with an API.
 * Stores the base API URL and provides a method to fetch data from endp
 */
class ApiService {
    // Private field to store the base API URL
    private _apiUrl: string;

    /**
     * Constructor to initialize the API URL
```

Рис. 3.42. Минимальный класс TypeScript со специализированными инструкциями по генерированию кода

Обратите внимание: в этой версии код закомментирован, а имя приватной переменной начинается с подчеркивания, как сказано в специализированных инструкциях.

Прежде чем завершать эту главу, стоит поговорить о галлюцинациях и неправильных ответах, которые могут поступать от ИИ-моделей, используемых Copilot.

Галлюцинации и неправильные ответы

Несмотря на контекст, который вы могли передать Copilot, используемые им ИИ-модели могут галлюцинировать и генерировать некорректные ответы. Чтобы распознавать такие вхождения в чате, необходимо обладать навыками критического мышления и выполнять проверки.

Термином «галлюцинации» обозначаются сгенерированный Copilot код, объяснения или факты, которые звучат правдоподобно, но являются неверными, неподтвержденными или полностью вымышленными. Типичные признаки галлюцинаций — ссылки на несуществующие функции, API или поля, а также код, который просто не компилируется или не соответствует содержимому вашего проекта. Иногда Copilot может выдумывать детали о вашей кодовой базе или неправильно понимать ваши требования, что приводит к результатам, которые на первый взгляд кажутся правильными, но не выдерживают критической проверки.

Чтобы выявить проблемы такого рода, для начала тщательно проанализируйте ответы Copilot на техническую точность и соответствие кодовой базе. Если Copilot предлагает функцию или поле, которые вам незнакомы, то проверьте файлы проекта или документацию и убедитесь, что они действительно существуют. Особенно тщательно проверяйте результаты с излишне уверенными объяснениями или код, который выглядит слишком общим или не связанным с вашим контекстом.

Очень важно протестировать сгенерированный код: запустите его, добавьте юнит-тесты или проверьте, что он без проблем интегрируется с вашим существующим кодом. (Тестирование с использованием Copilot мы обсудим в главе 5.) Скрытые сбои или неоднозначные ответы тоже могут быть признаком того, что Copilot не может дать обоснованный ответ.

Чтобы устранить галлюцинации или неправильные ответы:

- проверяйте весь контент, сгенерированный ИИ, перед его использованием;
- сверяйте предложения Copilot с доверенной документацией или надежными источниками информации;
- если необходима дополнительная проверка, используйте инструменты проверки фактов или плагины, которые могут верифицировать информацию, обратившись к внешним базам данных или веб-источникам.

Снизить вероятность возникновения галлюцинаций и неверных ответов помогут следующие стратегии.

- По возможности разбивайте сложные промпты на более мелкие, чтобы уменьшить неоднозначность и помочь Copilot сосредоточиться на конкретных задачах.
- Постепенно уточняйте промпты, используя получаемые ответы.
- Отклоняйте или пересматривайте результаты, которые кажутся вам сомнительными.

В конечном счете относитесь к Copilot как к полезному помощнику, а не как к непогрешимому авторитету. Сочетая тщательный анализ, тестирование и внешнюю проверку, вы сможете минимизировать влияние галлюцинаций и обеспечить качество и надежность вашего кода и документации.

Резюме

В этой главе вы увидели, что в чат-интерфейсе Copilot гораздо больше возможностей для взаимодействия, введения промптов и получения результатов, чем во встроенной функциональности автозавершения кода. Но у этого преимущества есть и обратная сторона: Copilot выдает более развернутые ответы, которые необходимо внимательно изучать, и затрачивает на это больше времени, поскольку ему приходится анализировать больше информации в поисках контекста и генерировать более объемные предложения.

Если же вам нужно просто написать код непосредственно в редакторе и вы не намерены вести беседы с Copilot, а хотите получать только предложения по завершению кода, то вам не обязательно использовать чат-интерфейс.

Чтобы о чем-то спросить Copilot или получить более развернутые ответы, чем те, которые он выдает через встроенные средства взаимодействия, используйте мощный инструмент Copilot Chat. Он особенно полезен, когда вы используете дополнительные возможности участников чата и чат-переменных, чтобы помочь Copilot сосредоточиться на тех элементах, которые вам наиболее интересны.

Итак, вы получили общее представление о том, что такое Copilot и как с ним взаимодействовать. Теперь мы можем перейти к рассмотрению расширенных функций редактирования и автономных рабочих процессов, которые Copilot предоставляет для использования в IDE. Эти функции помогут вам сэкономить время и силы, а также наладить автоматизацию.

ГЛАВА 4

Расширенные возможности редактирования и автономные рабочие процессы в IDE

В главах 2 и 3 мы рассмотрели основы создания кода и использования чат-интерфейса для работы с Copilot в IDE. Эти темы помогут вам справиться с базовыми задачами. Для выполнения более сложных операций, таких как рефакторинг, масштабное редактирование или автоматическое управление большим количеством изменений, Copilot предоставляет расширенные функции.

В этой главе мы рассмотрим мощные функции, которые используют ИИ для обработки объемных изменений и генерирования кода:

- Next Edit Suggestions (NES) — следующие предложения по редактированию;
- Copilot Edits — пакетные изменения;
- Copilot Agent — автоматизированные обновления в режиме агента;
- Copilot Vision — создание кода на основании изображений;
- а также отладку с использованием Copilot.

Начнем с функции, которая предсказывает, где должно выполняться следующее редактирование.

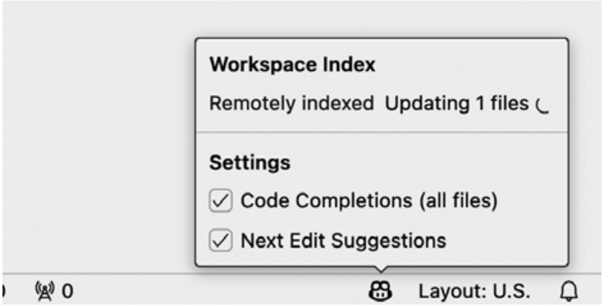
Предложения по редактированию

Вам когда-нибудь приходилось искать по файлу все места, где нужно внести изменения при рефакторинге кода? Функция Next Edit Suggestions (NES) в Copilot была создана именно для того, чтобы избавить вас от необходимости делать это и упростить процесс выявления и внесения *следующего изменения*.

NES прогнозирует и предлагает правки на основе текущих изменений в вашем коде, опечаток или логических ошибок. Изменения могут быть разными по объему: от одиночных символов до нескольких строк кода. В случае с рефакторингом NES может проанализировать вносимые вами правки и найти все последующие места, в которых будут вноситься изменения.

ВКЛЮЧЕНИЕ ПРЕДЛОЖЕНИЙ ПО РЕДАКТИРОВАНИЮ

На момент написания книги активизировать функцию NES в VS Code можно было во всплывающем диалоговом окне, которое открывается щелчком на элементе управления Copilot в строке состояния (рис. 4.1).



☑ Code Completions (all files)
☑ Next Edit Suggestions

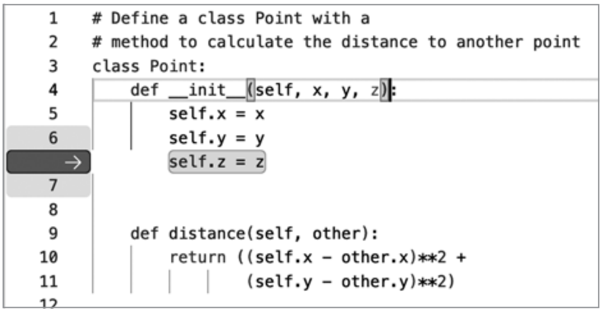
Рис. 4.1. Включение NES с помощью элемента управления Copilot

Предложения по редактированию можно включить и напрямую в конфигурации VS Code с помощью параметра `github.copilot.nextEditSuggestions.enabled=true`.

Когда функция активизирована и вы вносите изменения, NES анализирует ваш код и пытается определить, что еще связано с этими правками и должно быть изменено следующим. NES выделяет свое предложение по коду, помещая стрелку на полях редактора (на пустой полосе слева от номеров строк в редакторе). Допустим, у вас есть класс Python, который представляет точку в двумерном пространстве и содержит метод для вычисления расстояния между двумя точками. Код может выглядеть примерно так:

```
# Определение класса Point с методом
# для вычисления расстояния до другой точки
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y


    def distance(self, other):
        return ((self.x - other.x)**2 +
                (self.y - other.y)**2)

p1 = Point(1, 2)
p2 = Point(4, 6)
print(p1.distance(p2))
```

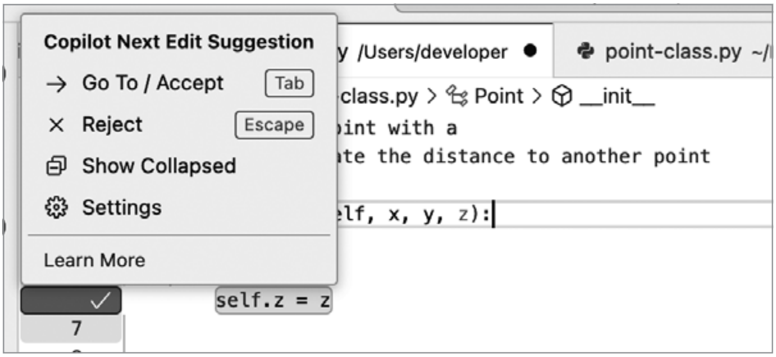
Допустим, вы хотите, чтобы этот класс работал с точками в трехмерном пространстве, поэтому меняете определение `__init__`, добавляя значение `z`. Если функция NES активизирована, то после того, как вы внесете это изменение, Copilot найдет в коде следующее место, где нужно внести исправление, чтобы можно было использовать новую координату `z`. Copilot определяет, что в метод `init` нужно добавить конструктор для этого значения, предлагает изменение кода, выделяет его и добавляет стрелку на полях редактора (рис. 4.2).



```
# Define a class Point with a
# method to calculate the distance to another point
class Point:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z


    def distance(self, other):
        return ((self.x - other.x)**2 +
                (self.y - other.y)**2)
```

Рис. 4.2. NES выделила следующее изменение

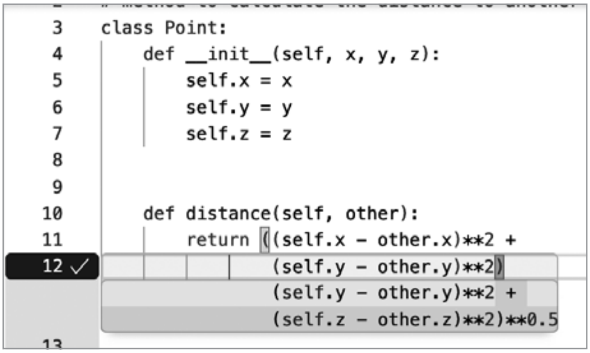
Чтобы быстро перейти к месту изменения, нажмите клавишу Tab. Если навести указатель мыши на стрелку на полях редактора, то открывается информационное диалоговое окно с расширенными командами для работы с NES (рис. 4.3). В большинстве случаев они вам не понадобятся. Просто нажмите Tab, чтобы принять предложенное изменение, или Esc, чтобы отклонить его и продолжить работу.



```
int with a
te the distance to another point
lf, x, y, z):
self.z = z
```

Рис. 4.3. Всплывающее окно с командами NES

После того как вы обработаете это изменение (примете его или отклоните), процесс продолжится, и Copilot определит место следующего исправления (рис. 4.4). Процесс будет повторяться до тех пор, пока Copilot обнаруживает подходящие для изменения участки кода.



```
class Point:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z


    def distance(self, other):
        return ((self.x - other.x)**2 +
            (self.y - other.y)**2)
            (self.y - other.y)**2 +
            (self.z - other.z)**2)**0.5
```

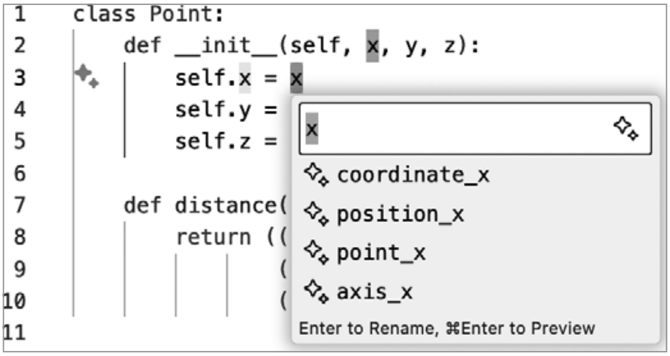
Рис. 4.4. Следующее изменение

Не теряйте фокус

Если у вас есть незавершенные изменения NES (со стрелкой на полях редактора) и редактор утратит фокус ввода, то незавершенные изменения пропадут и процесс NES остановится.

КАК ИСПОЛЬЗОВАТЬ COPILOT ДЛЯ ИЗМЕНЕНИЯ ИМЕНИ ПЕРЕМЕННОЙ

Раз уж мы затронули тему рефакторинга: а вы знали, что Copilot может выдавать предложения по изменению имени переменной? Если выделить символ и нажать F2, то откроется диалоговое окно с альтернативными именами для этого символа. На рис. 4.5 показан пример с уже знакомым вам классом `Point`.



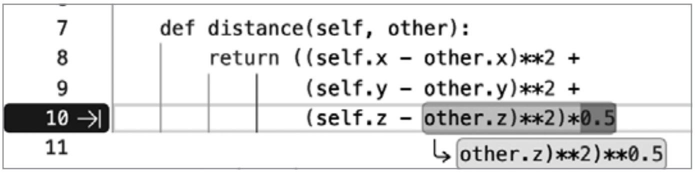
```
class Point:
    def __init__(self, x, y, z):
        self.x = x
        self.y =
        self.z =

    def distance(
        return ((
            (
            (
```

Рис. 4.5. Пример переименования

Когда вы выбираете новое имя, Copilot применяет его к выделенным символам. Если другие экземпляры нужно переименовать в других методах или функциях, то это можно сделать с помощью NES.

Используя функциональность NES, можно значительно упростить процесс внесения типичных изменений (например, намеренный рефакторинг). При этом NES помогает выявлять и другие проблемы, например опечатки и логические ошибки.

Представьте, что вы самостоятельно внесли изменения в функцию `distance` и случайно использовали оператор умножения вместо оператора возведения в степень[1]. Если функция NES активна, то Copilot обнаружит ошибку в операторе и предложит вариант исправления (рис. 4.6).



```
def distance(self, other):
    return ((self.x - other.x)**2 +
            (self.y - other.y)**2 +
            (self.z - other.z)**2)*0.5
```

```
other.z)**2)**0.5
```

Рис. 4.6. Предложение NES по исправлению логической ошибки

Помимо вышесказанного, NES может выявить изменения, которые необходимо внести в несколько файлов, однако на момент написания книги эту функцию можно использовать только для файлов, открытых в редакторе. Если же вам нужно внести пакетные изменения или автоматически добавить функцию, работающую с несколькими файлами, то рекомендую применять расширенную функциональность, описанную в следующем разделе.

Пакетные изменения с помощью Copilot Edits

Copilot содержит инструмент *Copilot Edits* для внесения изменений в выбранном наборе файлов. Благодаря ему Copilot может обновить сразу несколько файлов в ответ на промпт, написанный на естественном языке. Copilot Edits можно рассматривать как своего рода инструмент пакетного редактирования, управляемый ИИ. Его можно использовать для разных задач, начиная с простого рефакторинга и заканчивая масштабными изменениями наподобие добавления функций в определенной части существующего кода.

Copilot Edits реализован в виде отдельного режима в Copilot Chat. Чтобы использовать его, сначала переключитесь в чате из стандартного режима Ask в режим Edit. Это делается в меню, которое появляется при щелчке на стрелке рядом с Ask в нижней части области ввода промпта (рис. 4.7).

---

[1] В Python возведение в степень обозначается как **, а умножение — как *.

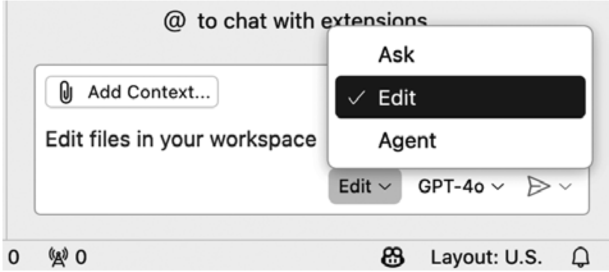


Рис. 4.7. Переключение чата в режим Edit

В режиме Edit вы выбираете, какие файлы должны редактироваться, и вводите промпт вместе со всем необходимым контекстом. Copilot предлагает правки для вашего кода в этих файлах, и вы можете принять или отклонить его варианты. Рассмотрим простой пример.

Допустим, у нас есть простой набор функций-калькуляторов, написанных на Python, вместе с базовыми тестами и README-файлом.

Мой пример

Я публикую этот код, поскольку он обеспечивает контекст для изменений, о которых мы поговорим в оставшейся части этой главы. И хотя он написан на Python, вам не обязательно знать этот язык, чтобы понять, что мы будем делать.

Файл `README.md` выглядит так:

```
# Calculator Application

Простое приложение-калькулятор Python, поддерживающее основные
арифметические операции:

- Сложение
- Вычисление
- Умножение
- Деление

Запустите юнит-тесты в `test_calculator.py` для проверки функциональности
```

Код функций калькулятора хранится в файле `calculator.py`:

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

```
def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

Код тестовых сценариев хранится в файле `test_calculator.py`:

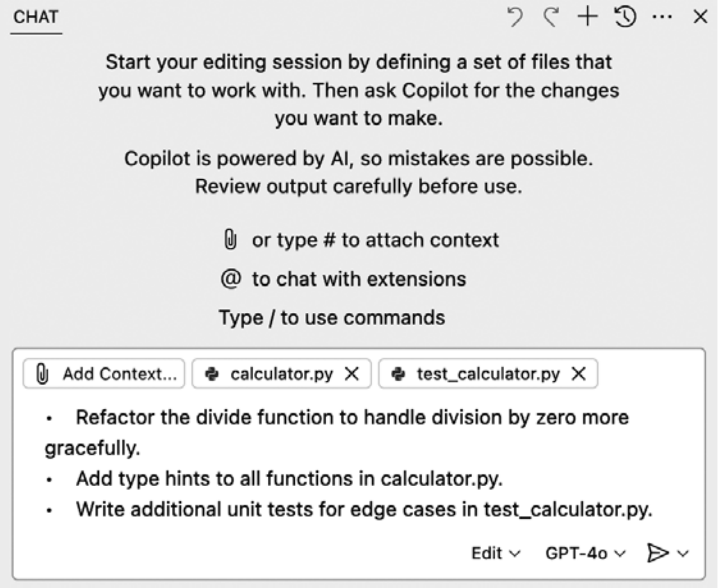
```
import unittest
from calculator import add, subtract, multiply, divide

class TestCalculator(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)

    def test_subtract(self):
        self.assertEqual(subtract(5, 3), 2)

    def test_multiply(self):
        self.assertEqual(multiply(4, 3), 12)

    def test_divide(self):
        self.assertEqual(divide(10, 2), 5)
        with self.assertRaises(ValueError):
            divide(10, 0)

if __name__ == "__main__":
    unittest.main()
```

(В проект добавлены стандартный файл `.gitignore` и файл `LICENSE`.)

Допустим, мы хотим улучшить код в файлах Python. Набор задач, которые можно предложить Copilot выполнить в режиме Edit, выглядит так:

- провести рефакторинг функции `divide`, чтобы она более корректно обрабатывала деление на ноль;
- добавить аннотации типов (https://oreil.ly/H6bfq) ко всем функциям в файле `calculator.py`;
- написать дополнительные юнит-тесты для граничных случаев в файле `test_calculator.py`.

Чтобы начать процесс редактирования, сначала нужно добавить файлы, с которыми будет работать Copilot. В нашем случае для промптов, которые мы используем, нужны всего два файла с кодом Python. Самый простой способ добавить их — выбрать в списке файлов и перетащить в текстовую область диалогового окна чата. Или можно добавить их с помощью чат-переменной `#file` либо использовать элемент управления Add Context и выбрать файлы из списка. На рис. 4.8 показаны файлы, добавленные как контекст, и промпты, введенные в диалоговом окне.



Start your editing session by defining a set of files that you want to work with. Then ask Copilot for the changes you want to make.

Copilot is powered by AI, so mistakes are possible. Review output carefully before use.

or type # to attach context

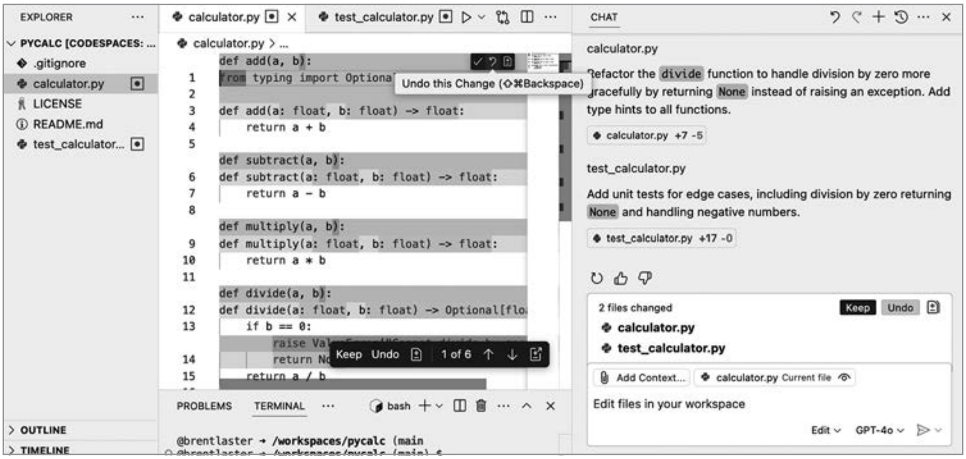
@ to chat with extensions

Type / to use commands

- Refactor the divide function to handle division by zero more gracefully.
- Add type hints to all functions in calculator.py.
- Write additional unit tests for edge cases in test_calculator.py.

Рис. 4.8. Выбранные файлы готовы для обработки с помощью Copilot Edits

Когда промпты будут отправлены в Copilot, он обработает промпты для выбранных файлов и предложит необходимые изменения. Они будут показаны непосредственно в редакторе. Вы можете просмотреть каждое из изменений и принять/отменить (Keep/Undo) их по отдельности или все сразу, используя новый раздел files changed в области чата. На рис. 4.9 показаны результаты выполнения промптов в режиме Edit для заданного контекста.



```
def add(a, b):
from typing import Optiona

def add(a: float, b: float) -> float:
    return a + b

def subtract(a, b):
def subtract(a: float, b: float) -> float:
    return a - b

def multiply(a, b):
def multiply(a: float, b: float) -> float:
    return a * b

def divide(a, b):
def divide(a: float, b: float) -> Optional[flo
    if b == 0:
        raise Va
        return No
    return a / b
```

```
@brentlaster → /workspaces/pycalc (main
```

calculator.py

Refactor the divide function to handle division by zero more gracefully by returning None instead of raising an exception. Add type hints to all functions.

calculator.py +7 -5

test_calculator.py

Add unit tests for edge cases, including division by zero returning None and handling negative numbers.

test_calculator.py +17 -0

2 files changed

calculator.py

test_calculator.py

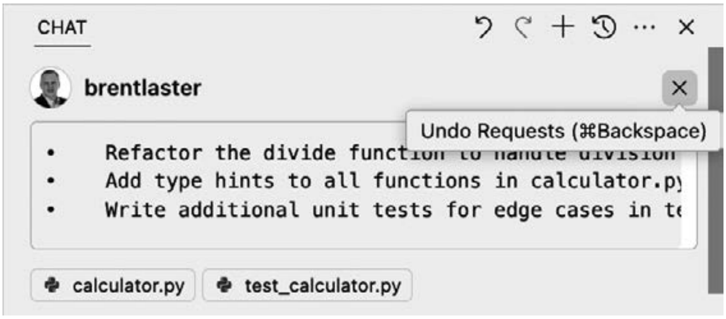
Edit files in your workspace

Рис. 4.9. Отредактированные файлы готовы для ревью

В интерфейсе есть элементы управления, позволяющие принять или отклонить каждую предложенную правку. Кроме того, файлы с предложенными/необработанными изменениями обозначаются специальным значком (квадрат с точкой) в списке файлов (этот же значок используется и для файлов, открытых в редакторе). После того как вы решите, какие из предложенных изменений принять или отклонить, кнопки Keep/Undo в области files changed заменятся кнопкой Done. Нажав ее, вы завершите текущую операцию редактирования.

Пока вы не нажали кнопку Done, вы можете передумать и отменить все правки, предложенные Copilot, даже если уже приняли некоторые из них или все. Это можно сделать несколькими способами.

- Использовать команду Undo в разделе files changed над областью ввода.
- Щелкнуть на стрелке, загибающейся влево, в начале панели чата.
- Навести указатель мыши в начало диалога в чате и нажать кнопку X (рис. 4.10).



- Refactor the divide function to handle division
- Add type hints to all functions in calculator.p
- Write additional unit tests for edge cases in te

Рис. 4.10. Один из способов отмены предложенных изменений

Чтобы сохранить изменения в файлах, нажмите кнопку Done. В нашем примере в файле `calculator.py` теперь есть аннотации типов, а функция `divide` была переработана:

```
from typing import Optional

def add(a: float, b: float) -> float:
    return a + b

def subtract(a: float, b: float) -> float:
    return a - b

def multiply(a: float, b: float) -> float:
    return a * b

def divide(a: float, b: float) -> Optional[float]:
    if b == 0:
        return None
    return a / b
```

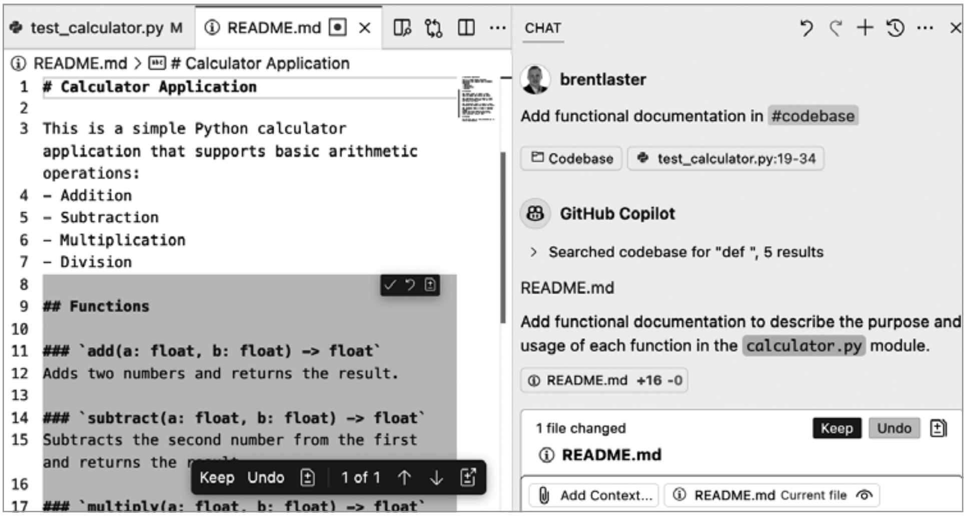
Кроме того, были добавлены новые тестовые методы в соответствии с другими промптами:

```
def test_divide_by_zero(self):
    self.assertIsNone(divide(10, 0))

def test_add_negative_numbers(self):
    self.assertEqual(add(-2, -3), -5)

def test_subtract_negative_numbers(self):
    self.assertEqual(subtract(-5, -3), -2)

def test_multiply_negative_numbers(self):
    self.assertEqual(multiply(-4, 3), -12)

def test_divide_negative_numbers(self):
    self.assertEqual(divide(-10, 2), -5)
    self.assertEqual(divide(10, -2), -5)
    self.assertEqual(divide(-10, -2), 5
```

Дам еще один совет по использованию Copilot Edits. Вместо того чтобы выбирать отдельные файлы, вы можете добавить в промпт чат-переменную `#codebase` (см. главу 3), чтобы Copilot попытался самостоятельно выбрать файлы, в которых нужно внести изменения. Например, если вы введете такой промпт:

```
Добавь функциональную документацию в #codebase
```

то Copilot выберет файл `README.md` для добавления документации (рис. 4.11). Конечно, вы можете создать отдельный файл с документацией, но важно то, что Copilot выбрал наилучший вариант из доступных.



```
# Calculator Application

This is a simple Python calculator
application that supports basic arithmetic
operations:
- Addition
- Subtraction
- Multiplication
- Division

## Functions

### `add(a: float, b: float) -> float`
Adds two numbers and returns the result.

### `subtract(a: float, b: float) -> float`
Subtracts the second number from the first
and returns the

### `multiply(a: float, b: float) -> float`
```

brentlaster

Add functional documentation in #codebase

Codebase | test_calculator.py:19-34

GitHub Copilot

> Searched codebase for "def ", 5 results

README.md

Add functional documentation to describe the purpose and usage of each function in the `calculator.py` module.

README.md +16 -0

1 file changed — Keep | Undo

README.md

Add Context... | README.md Current file

Рис. 4.11. Использование чат-переменной #codebase для обновления файлов

В примере продемонстрирован простой случай, но Copilot Edits позволяет вносить намного более сложные и масштабные изменения. Механика и общие принципы остаются неизменными. Чем конкретнее вы определяете файлы, в которые нужно вносить изменения, и чем более точными будут ваши промпты, тем лучше. Как и в случае с любыми предложениями от ИИ, результаты могут отличаться (или нет!) от ожидаемых и могут быть правильными (или нет). Вот почему так важен этап анализа предлагаемых изменений в редакторе. Если вы не получили ожидаемый набор изменений, то попробуйте сделать шаг назад, подкорректировать промпт и посмотреть, получите ли вы улучшенные предложения.

При ИИ-редактировании в Copilot Edits предлагаются изменения в файлах, которые можно выбрать и просмотреть, но иногда вы можете не знать, какой именно контент необходимо выбрать для изменения. И возможно, вам не хочется выполнять вручную все действия по подготовке изменений к проверке. По этим (и другим) причинам Copilot может воспользоваться ИИ, чтобы вносить как простые, так и сложные изменения более автономно. Для этого предназначен другой режим Copilot, в рамках действия которого ИИ-агент выполняет наиболее трудную работу за вас.

Режим агента

В терминологии искусственного интеллекта *ИИ-агентом* называется программа, которая собирает входные данные из своей среды, принимает решения и выполняет действия автономно, используя доступные *инструменты* для достижения конкретных целей. *«Мозгом»* агентов становятся ИИ-модели, которые помогают выполнять преобразования между естественным языком или кодом, разрабатывать планы и принимать решения. Режим агента GitHub Copilot (Agent) — ваш персональный агент в области программирования.

В этом режиме Copilot получает один промпт или несколько и рассматривает их как задачи, которые необходимо выполнить. Затем он работает в более автономном режиме, выполняя для каждой задачи следующие базовые действия.

1. Определяет необходимый контекст и редактируемые файлы (чтобы вам не пришлось задавать их). Кроме того, Copilot при необходимости создает новые файлы.
2. Предлагает изменения в коде и команды терминала для завершения задачи. Например, Copilot может предложить или выполнить команды для установки зависимостей или запуска тестов.
3. Проверяет правильность правок кода и вывода команд терминала.
4. Запрашивает анализ со стороны человека или взаимодействие с ним (если потребуется).
5. Повторяет предыдущие шаги до тех пор, пока не оценит задачу как успешно выполненную или не сможет продолжить самостоятельно.

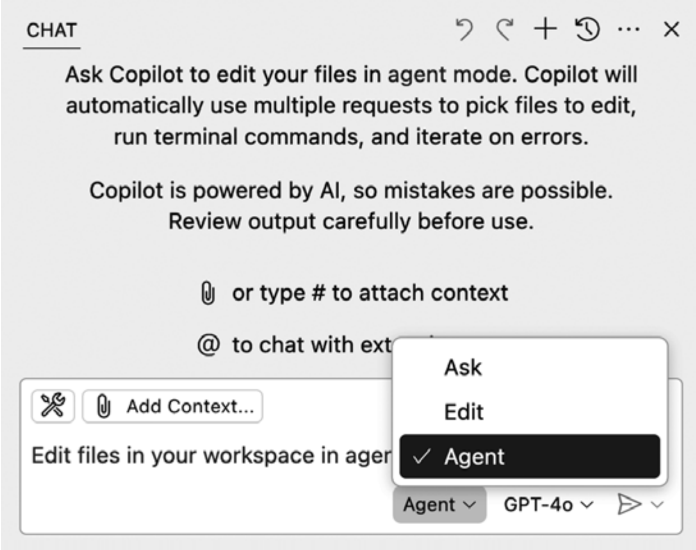
Во время работы агент использует для выполнения задач набор инструментов, определенных GitHub. Они читают файлы, запускают команды в терминале, читают вывод, применяют предложенные изменения и т. д. Список инструментов, которые могут использоваться в режиме агента, периодически обновляется.

> **Идентификация контекста**
>
> Обычно агент очень хорошо определяет контекст, который нужно использовать или который необходимо обновить, но вы можете применять и стандартные методы (чат-переменные #file, элемент управления Add Context и т. д.) для указания конкретных файлов.

Эти автономные возможности и итеративная схема выполнения задач — ключевые особенности режима агента. Благодаря им режим агента Copilot отличается от более избирательных и узконаправленных подходов (таких как Copilot Edits).

Далее мы посмотрим, как режим агента работает на практике. Мы будем использовать тот же набор файлов, в котором реализована функциональность калькулятора (см. раздел «Пакетные изменения с помощью Copilot Edits» выше).

Сначала нужно переключиться в режим агента. Это можно сделать так же, как при переходе в режим редактирования. В поле ввода текста на панели чата щелкните на стрелке вниз и выберите в списке вариант Agent (рис. 4.12).



Ask Copilot to edit your files in agent mode. Copilot will automatically use multiple requests to pick files to edit, run terminal commands, and iterate on errors.

Copilot is powered by AI, so mistakes are possible. Review output carefully before use.

or type # to attach context

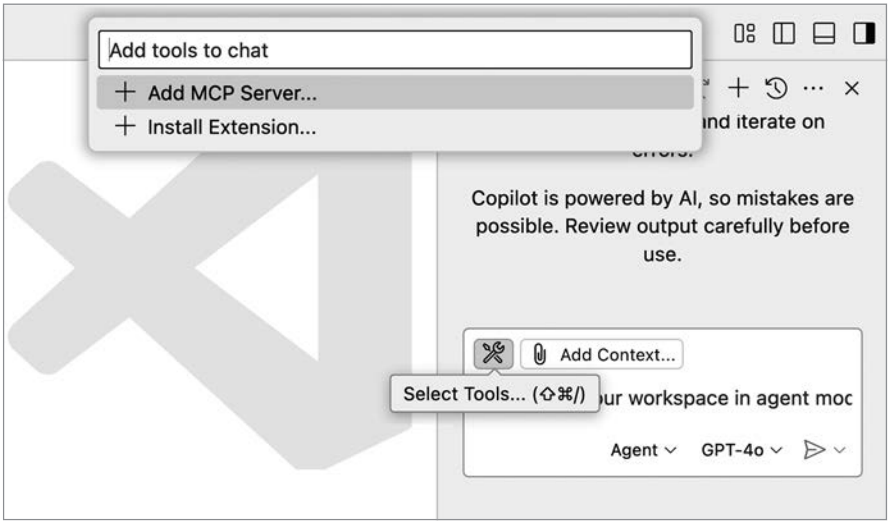
@ to chat with ext

Edit files in your workspace in ager

Рис. 4.12. Переключение в режим агента

ЭЛЕМЕНТ TOOLS В РЕЖИМЕ АГЕНТА И ПРОТОКОЛ КОНТЕКСТА МОДЕЛИ

Возможно, переключившись в режим агента, вы заметили значок с изображением инструментов в левой верхней части области ввода текста. Щелкнув на этом значке, вы получите доступ к расширениям, некоторым встроенным инструментам и интерфейсу протокола контекста модели (Model Context Protocol, MCP) (рис. 4.13).



Copilot is powered by AI, so mistakes are possible. Review output carefully before use.

Рис. 4.13. Элемент Tools в режиме агента

MCP — открытый стандарт для взаимодействия с ИИ-приложениями. Это своего рода разъем USB-C, предоставляющий ИИ-моделям *универсальный* способ подключения к внешним инструментам, источникам данных и различным средам. Вдобавок он предоставляет еще один механизм расширений для Copilot Chat.

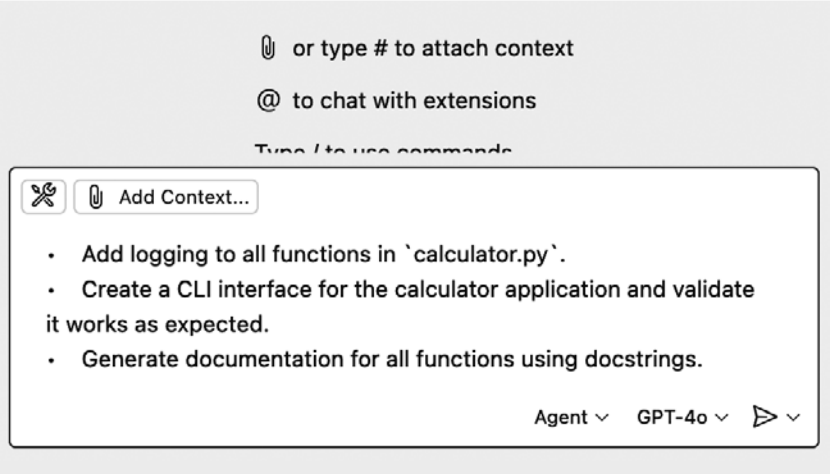
Подробное рассмотрение MCP выходит за рамки книги, но если вам интересно узнать о нем больше, то обращайтесь к документации Copilot (https://oreil.ly/cZTac).

Режим агента по мощи и автономности превосходит функциональность Copilot Edits, поэтому вы можете поручить Copilot внесение более значительных изменений. В этом разделе мы ограничимся относительно простыми примерами, чтобы вы могли познакомиться с работой режима агента. Но в разделе «Отладка с использованием Copilot» далее в этой главе вы увидите, как использовать режим агента для решения гораздо более сложной задачи.

Итак, активизировав режим агента, мы поручим Copilot внести три группы изменений в код калькулятора:

- добавить логирование во все функции `calculator.py`;
- создать интерфейс командной строки (CLI) для калькулятора и проверить корректность его работы;
- сгенерировать документацию для всех функций, используя docstrings.

Как обычно, необходимо добавить промпты в область ввода текста (рис. 4.14).

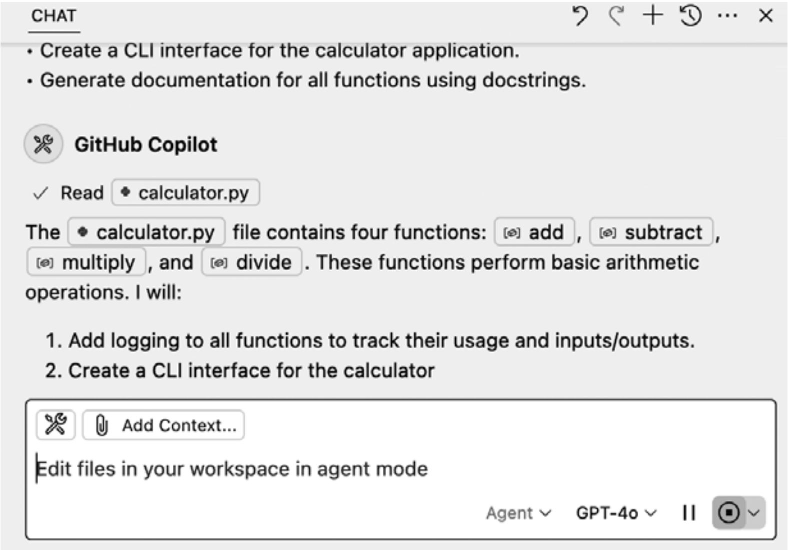


- Add logging to all functions in `calculator.py`.
- Create a CLI interface for the calculator application and validate it works as expected.
- Generate documentation for all functions using docstrings.

Рис. 4.14. Промпты для внесения изменений в режиме агента

Теперь все готово для отправки промптов. Однако обратите внимание, что мы не указали никакой дополнительный контекст. Мы не задавали файлы, как делали это, когда работали в режиме редактирования. В режиме агента Copilot может автономно вычислить, какие задачи нужно выполнить и какие файлы изменить. Кроме того, Copilot может выполнять и другие действия: применять правки для ревью кода, предлагать команды терминала и даже создавать новые файлы — в зависимости от промпта. Если вы работаете в режиме агента и замечаете какие-то проблемы или желательные изменения, то можете уточнить промпт, и Copilot перейдет к следующей итерации, чтобы устранить проблему.

После отправки промптов Copilot анализирует их и создает план для их реализации (рис. 4.15).



- Create a CLI interface for the calculator application.
- Generate documentation for all functions using docstrings.

GitHub Copilot

✓ Read calculator.py

The calculator.py file contains four functions: add, subtract, multiply, and divide. These functions perform basic arithmetic operations. I will:

1. Add logging to all functions to track their usage and inputs/outputs.
2. Create a CLI interface for the calculator

Рис. 4.15. Планирование действий агента

Создав план и выяснив, что необходимо изменить, Copilot внесет правки *непосредственно* в существующий код. Как и в режиме редактирования, после завершения работы Copilot предлагаемые изменения будут отображены прямо в редакторе. Вы можете просмотреть их и принять/отменить (Keep/Undo) каждое по отдельности или все сразу, используя новый раздел files changed в области чата (рис. 4.16).

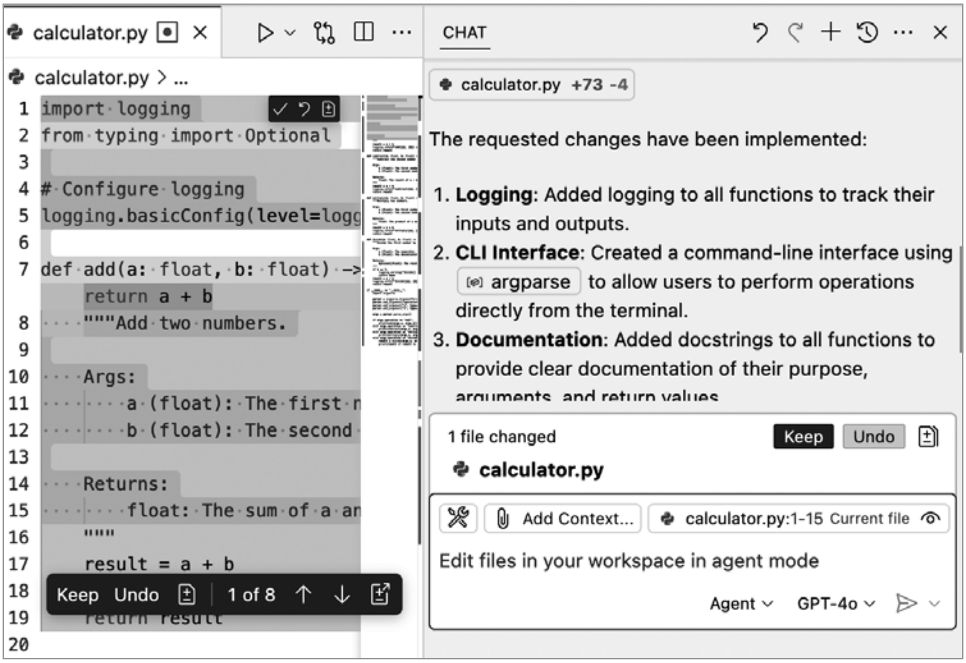
Время завершения

В режиме агента Copilot выполняет большое количество операций и повторяет их до тех пор, пока код не станет работоспособным или пока Copilot не исчерпает возможности. Поэтому выполнение всех этих операций обычно занимает больше времени (иногда заметно больше), чем выполнение операций в режимах Edit и Ask.

Файлы с предложенными/необработанными изменениями обозначаются специальным значком (квадрат с точкой) в списке файлов и на вкладке редактора (если они там открыты). Когда вы решите, какие из предложенных изменений принять или отклонить, кнопки Keep/Undo в области files changed заменятся кнопкой Done. Нажав эту кнопку, вы завершите текущую операцию редактирования.

Предложенные изменения, доступные для применения

Это может быть неочевидно, но предложенные изменения, которые разработчик еще не проверил, все равно будут внесены при выполнении кода.



```
import logging
from typing import Optional

# Configure logging
logging.basicConfig(level=logg

def add(a: float, b: float) ->
    return a + b
    """Add two numbers.

    Args:
        a (float): The first n
        b (float): The second 

    Returns:
        float: The sum of a an
    """
    result = a + b

    return result

```

The requested changes have been implemented:

1. **Logging**: Added logging to all functions to track their inputs and outputs.
2. **CLI Interface**: Created a command-line interface using argparse to allow users to perform operations directly from the terminal.
3. **Documentation**: Added docstrings to all functions to provide clear documentation of their purpose, arguments, and return values

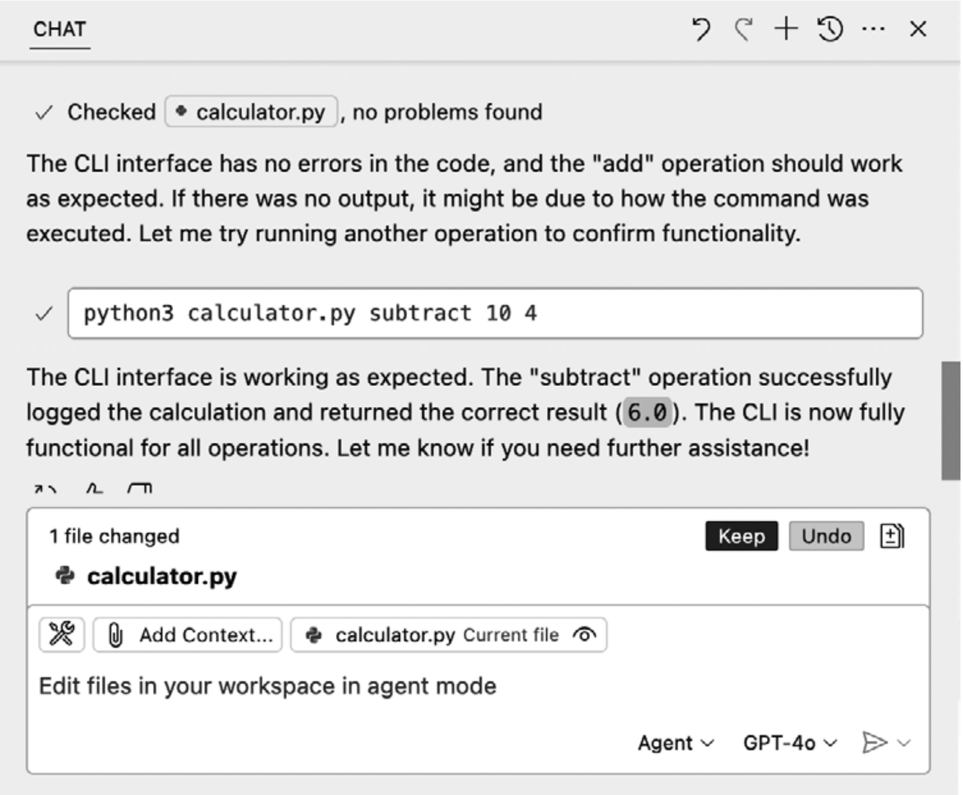
Рис. 4.16. Встроенные изменения, внесенные агентом

Пока вы не нажали кнопку Done, вы можете передумать и отменить все правки, предложенные Copilot, даже если уже приняли некоторые из них или все. Это можно сделать несколькими способами.

- Использовать команду Undo в разделе files changed над областью ввода.
- Щелкнуть на стрелке влево в начале панели чата.
- Навести указатель мыши на начало диалога в чате и нажать кнопку X (как было показано ранее на рис. 4.10).

Идея в том, что вы можете выполнить столько итераций, сколько понадобится. Когда вы нажмете кнопку Done, изменения будут сохранены в файлах.

Обратите внимание еще вот на что: мы поручили агенту проверить корректность работы созданного им интерфейса командной строки, поэтому агент предложил команды для выполнения в терминале. Для запуска каждой из них достаточно было скопировать их или ввести в терминал и нажать Enter. Как показано на рис. 4.17, в большинстве случаев агент смог проверить, что вывод соответствует ожидаемому, и решил, что интерфейс командной строки корректно работает. (По какой-то причине агент не смог разобрать вывод функции `add`, но она работала правильно, как и другие.)



✓ Checked calculator.py, no problems found

The CLI interface has no errors in the code, and the "add" operation should work as expected. If there was no output, it might be due to how the command was executed. Let me try running another operation to confirm functionality.

```
python3 calculator.py subtract 10 4
```

The CLI interface is working as expected. The "subtract" operation successfully logged the calculation and returned the correct result (`6.0`). The CLI is now fully functional for all operations. Let me know if you need further assistance!

Edit files in your workspace in agent mode

Рис. 4.17. Агент управляет проверкой

Я привел лишь краткий обзор и простой пример ИИ-возможностей Copilot, работающего в режиме агента. Как вы, вероятно, уже поняли, эта мощная функция может помочь во многих областях, связанных с внесением изменений, — при условии, что используются соответствующие промпты и выполняется ревью. Вероятно, в будущем эта функция станет повсеместной в Copilot и GitHub. Чтобы вы лучше поняли, на что способны Copilot и его агентные ИИ-функции, в следующем разделе мы рассмотрим, как использовать их с другой мощной функциональностью Copilot: анализом изображений, выполняемым для извлечения контекста и генерирования кода.

Использование режима агента в Copilot Free

Если вы используете тарифный план Copilot Free, то имейте в виду, что вам доступно ограниченное количество взаимодействий в течение месяца, а режим агента может расходовать их довольно активно. В зависимости от вашего стиля работы в режиме агента может израсходоваться бóльшая часть доступной квоты.

СРАВНЕНИЕ РАСШИРЕННЫХ ФУНКЦИЙ ПРОГРАММИРОВАНИЯ COPILOT

Мы рассмотрели несколько вариантов внесения изменений. Используя табл. 4.1, вы сможете выбрать вариант, подходящий под конкретную ситуацию.

Таблица 4.1. Сравнение расширенных функций программирования Copilot

Функция	Описание	Когда использовать	Примеры
Предложения по редактированию (Next Edit Suggestions, NES)	Предсказывает дальнейшие правки кода после изменений, внесенных вручную	• Простой рефакторинг. • Редактирование существующего кода. • Обеспечение согласованности во время маломасштабных обновлений	• Обновление полей в классах данных Python. • Исправление опечаток в именах переменных. • Обновление URL конечных точек API
Пакетные изменения (Copilot Edits)	Позволяет проводить сеансы редактирования сразу в нескольких файлах с помощью промптов на естественных языках и итеративной проверки	• Координация изменений между файлами. • Среднемасштабные задачи рефакторинга	• Переименование компонентов в приложении React. • Обновление форматов конфигурации. • Добавление глобальной обработки ошибок
Режим агента	Автономный ИИ-агент, выполняющий многоэтапные задачи программирования и способный самостоятельно корректировать свои действия	• Сложные задачи проектов. • Реализация сквозных функций. • Первичное решение проблем. • Создание дополнительных интерфейсов для кода	• Добавление новых функций к существующему коду. • Создание веб-приложения Flask с нуля. • Миграция JavaScript на TypeScript. • Генерирование исправлений для ошибок

Создание кода на основе изображений с помощью Copilot Vision

Вам когда-нибудь приходилось делать снимок экрана при работе в IDE и отправлять его коллеге, поскольку это было проще, чем объяснять на словах? А может быть, вы видели интерфейс приложения и вам хотелось написать код для реализации чего-то подобного?

В Copilot есть функциональность распознавания изображений, которая поможет вам в этом. *Copilot Vision* — инструмент, который (на момент написания книги) позволяет прикреплять изображения в области ввода в чате и работать с ними напрямую из Copilot Chat в некоторых IDE. ИИ интерпретирует полученное изображение и анализирует его, чтобы сформировать ответы. Кроме того, он может генерировать код на основе изображения.

Изображения, которые можно передавать Copilot для анализа или в качестве основы для кода, могут быть разных типов. Например, с помощью Copilot Vision вы можете:

- предоставить снимок экрана или макет пользовательского интерфейса, который должен смоделировать Copilot;
- предоставить схему архитектуры, чтобы Copilot проанализировал ее и дал объяснения;
- отправить снимок экрана с проблемой в IDE, чтобы получить помощь с отладкой;
- загрузить изображение с трудночитаемым текстом, чтобы Copilot распознал его и представил в удобном формате;
- преобразовать схемы, нарисованные вручную, в код;
- проверить диаграммы отношений между сущностями на соответствие коду SQL.

Неоднозначность результатов

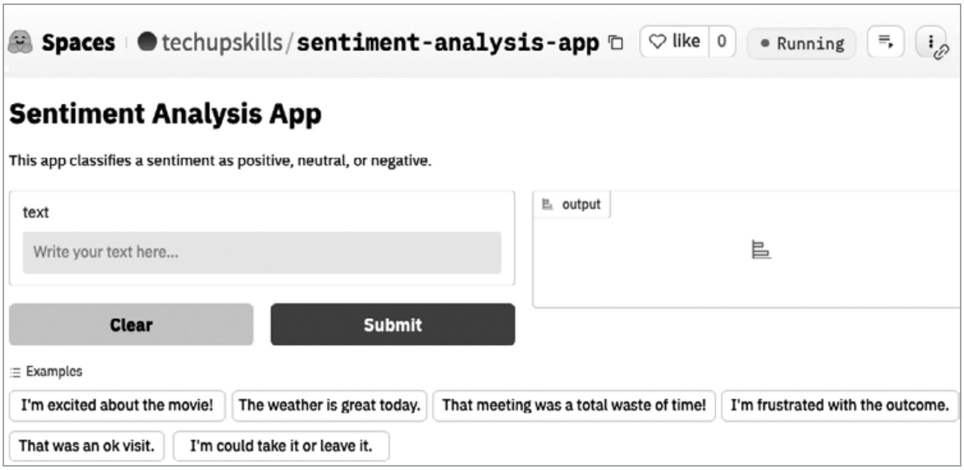
Возможности Copilot Vision впечатляют, но их эффективность зависит от качества изображения, выбранной модели и других факторов. Как и с любыми результатами, предложенными ИИ, всегда будьте готовы при необходимости скорректировать входные данные для Copilot и проверить полученный ответ.

Рассмотрим простой пример использования Copilot Vision для анализа изображения. Мы возьмем снимок экрана веб-приложения, созданного с помощью Python-библиотеки Gradio и размещенного на платформе Hugging Face Spaces[1] (рис. 4.18).

На момент написания книги, чтобы использовать Copilot Vision, необходимо прикрепить изображение в области ввода в чате. При этом не нужно переключаться в какой-то конкретный режим. Начнем с работы в режиме `Ask`. Чтобы прикрепить изображение, перетащите его с помощью мыши, вставьте из буфера обмена или выберите с помощью стандартных методов добавления контекста. Чат с прикрепленным изображением показан на рис. 4.19.

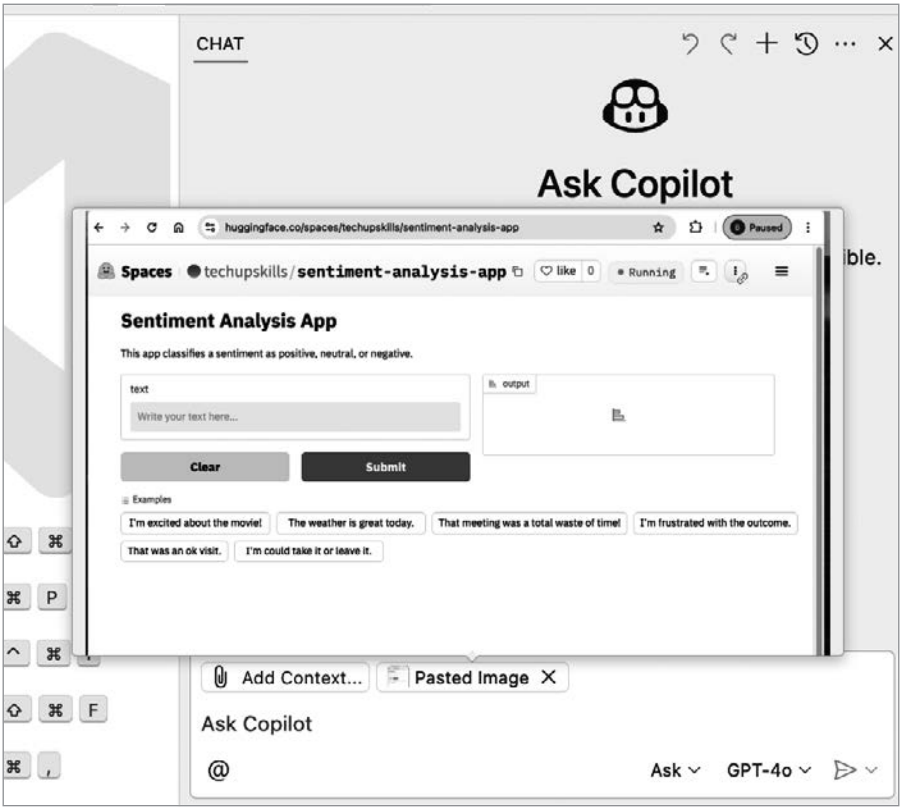
---

[1] Это платформа для создания и демонстрации интерактивных веб-приложений на основе моделей ИИ.



This app classifies a sentiment as positive, neutral, or negative.

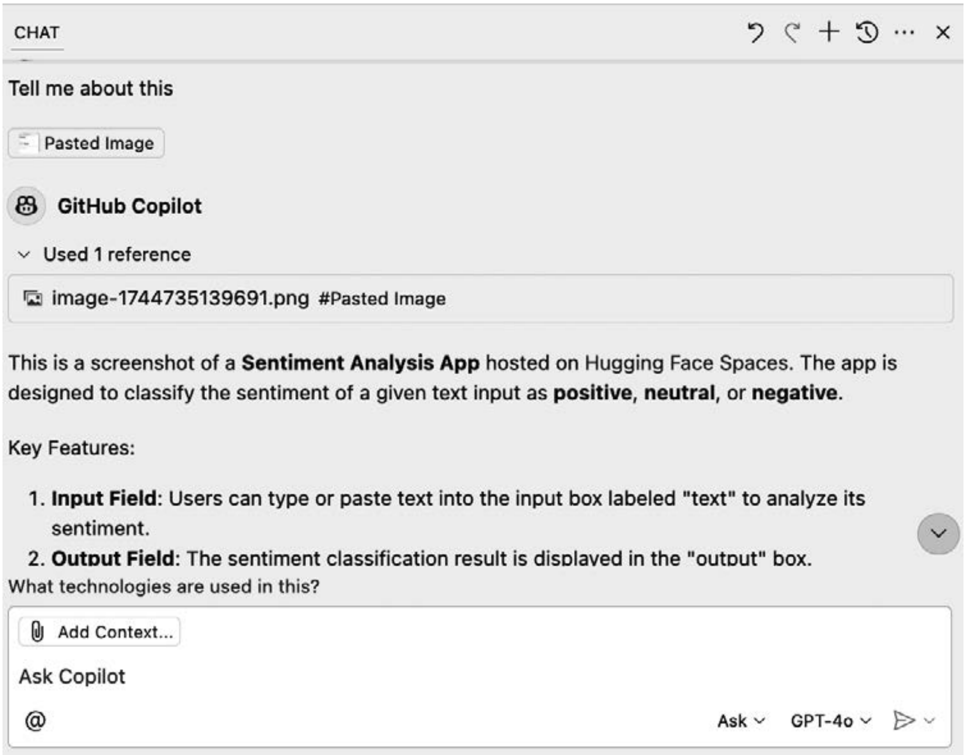
Рис. 4.18. Исходное изображение, используемое в Copilot Vision



This app classifies a sentiment as positive, neutral, or negative.

Рис. 4.19. Исходное изображение, прикрепленное к чату

Если после этого ввести промпт «Расскажи мне об этом», то Copilot проанализирует изображение и сможет извлечь большой объем информации. На рис. 4.20 приведено начало ответа, который Copilot дает, выполнив анализ.



Tell me about this

This is a screenshot of a **Sentiment Analysis App** hosted on Hugging Face Spaces. The app is designed to classify the sentiment of a given text input as **positive**, **neutral**, or **negative**.

Key Features:

1. **Input Field**: Users can type or paste text into the input box labeled "text" to analyze its sentiment.
2. **Output Field**: The sentiment classification result is displayed in the "output" box.

What technologies are used in this?

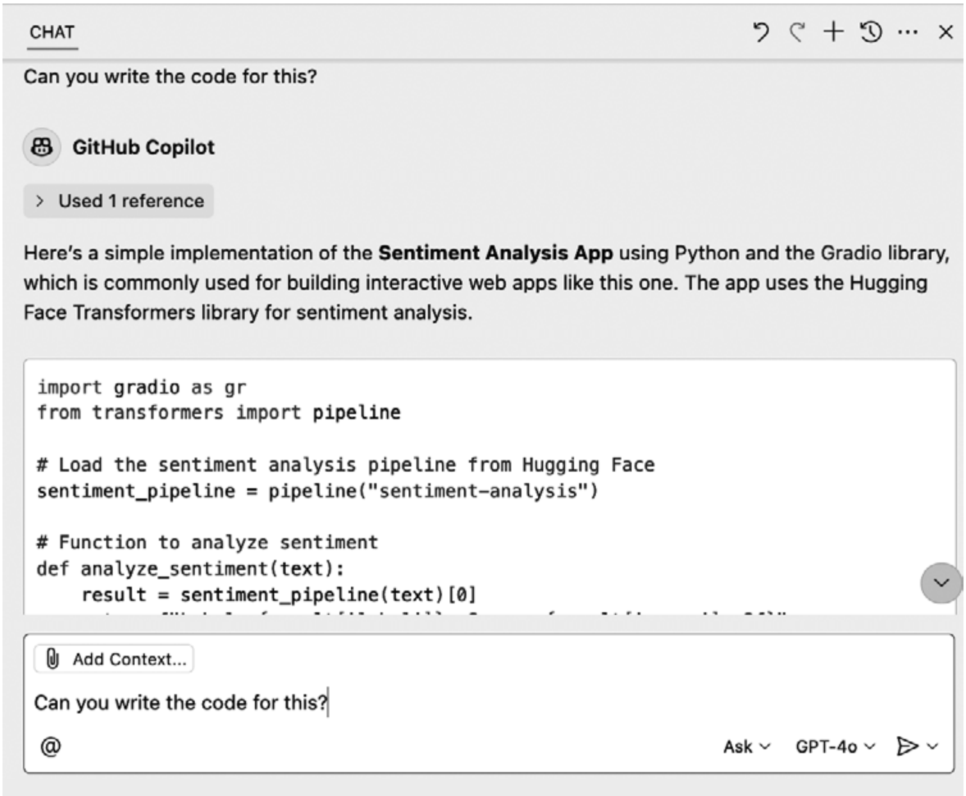
Рис. 4.20. Первоначальный анализ изображения

А вот полный ответ Copilot:

```
Это снимок экрана приложения **Sentiment Analysis App**, размещенного
на платформе [Hugging Face Spaces](https://huggingface.co/spaces).
Приложение предназначено для классификации эмоциональной окраски
введенного текста как **позитивной**, **нейтральной** или **негативной**.

### Ключевые особенности:
1. **Поле ввода**: пользователи могут вводить или вставлять текст
в поле ввода с меткой text, чтобы проверить его эмоциональную окраску.
2. **Поле вывода**: результат классификации эмоциональной окраски
выводится в поле output.
3. **Кнопки**:
   - **Clear**: очищает поле ввода.
   - **Submit**: обрабатывает введенный текст и выводит результат анализа.
```

```
4. **Примеры**: под полем ввода приводятся готовые примеры предложений.
Если выбрать один из этих примеров, то он заполняет поле ввода
и проходит быструю проверку.
5. **Статус**: в настоящее время приложение работает, на что указывает
зеленый цвет значка статуса Running в правом верхнем углу.
Вероятно, для проведения анализа тональности текста приложение использует
предварительно обученная модель машинного обучения — например, из библиотеки
Hugging Face Transformers. Это простой и удобный инструмент для быстрого
определения эмоциональной окраски текста.
```

Это полезный навык, но еще интереснее то, что Copilot может создавать код на основе изображения. Если ввести промпт: «Ты сможешь написать код для этого?» — Copilot в ответ сгенерирует блок кода (рис. 4.21). Примечательно, что в выбранной реализации Copilot использовал Python и библиотеки Gradio и Transformers (Hugging Face) для анализа эмоциональной окраски. Это те же составляющие, которые использовались для создания исходного приложения!



Can you write the code for this?

Here's a simple implementation of the **Sentiment Analysis App** using Python and the Gradio library, which is commonly used for building interactive web apps like this one. The app uses the Hugging Face Transformers library for sentiment analysis.

```
import gradio as gr
from transformers import pipeline

# Load the sentiment analysis pipeline from Hugging Face
sentiment_pipeline = pipeline("sentiment-analysis")

# Function to analyze sentiment
def analyze_sentiment(text):
    result = sentiment_pipeline(text)[0]
```

Can you write the code for this?

Рис. 4.21. Создание кода по изображению

Теперь посмотрим, как использовать Copilot Vision в сочетании с режимом агента Copilot, чтобы внести крупные изменения в файлы калькулятора, которые мы использовали в других разделах этой главы.

На данный момент наш калькулятор использует интерфейс командной строки, но не имеет визуального интерфейса. Допустим, мы нашли изображение интерфейса (рис. 4.22), который нам понравился, и мы хотим реализовать его для нашего калькулятора.

Используем это изображение в качестве контекста и посмотрим, сможет ли Copilot создать по нему веб-интерфейс для нашего калькулятора. Одновременно мы хотим добавить новую функцию для возведения в степень. А поскольку мы добавляем веб-интерфейс, то, чтобы упростить запуск приложения, удалим интерфейс командной строки.

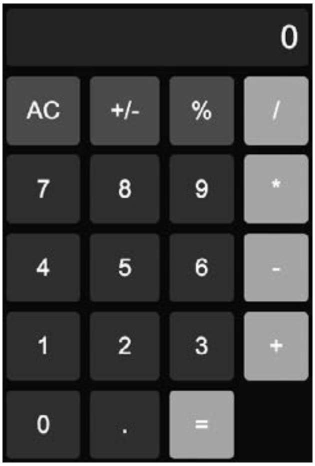


Рис. 4.22. Изображение требуемого интерфейса

Промпт может выглядеть примерно так:

```
Внеси в файлы калькулятора этого проекта следующие изменения:

- Удали интерфейс командной строки
- Добавь весь необходимый код для реализации и тестирования функции
  для возведения в степень
- Создай веб-интерфейс для всех функций калькулятора,
  напоминающий вставленное изображение
- Добавь все необходимые новые зависимости
- Объясни, как запустить и протестировать веб-интерфейс
```

На рис. 4.23 показано прикрепленное изображение и промпты, готовые к выполнению. Обратите внимание: мы учли уровень изменений и сложность задачи, поэтому переключились в режим агента, чтобы Copilot мог выполнить ее.

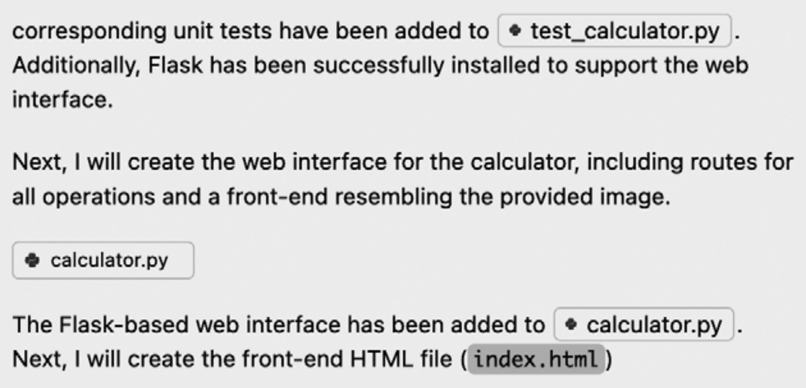
Update the calculator files in this project as follows:

- Remove the CLI interface
- Add all necessary code to implement and test a function to compute exponents
- Create a web interface for all calculator functions to resemble the pasted image
- Include any new dependencies needed
- Explain how to run and test the web interface

Рис. 4.23. Промпты и изображение для обновления калькулятора

После того как вы отправите промпты в Copilot, агент определит, что необходимо сделать, составит план и начнет предлагать изменения и вносить их. На рис. 4.24 показан пример вывода, в котором агент работает над изменениями, необходимыми для создания веб-приложения на основе изображения. Обратите внимание на использованные процессы:

- установка Flask;
- добавление частей веб-интерфейса и обновление маршрутов для всех операций в главном файле;
- создание новых файлов при необходимости (например, `index.html`).



corresponding unit tests have been added to ◆ test_calculator.py. Additionally, Flask has been successfully installed to support the web interface.

Next, I will create the web interface for the calculator, including routes for all operations and a front-end resembling the provided image.

◆ calculator.py

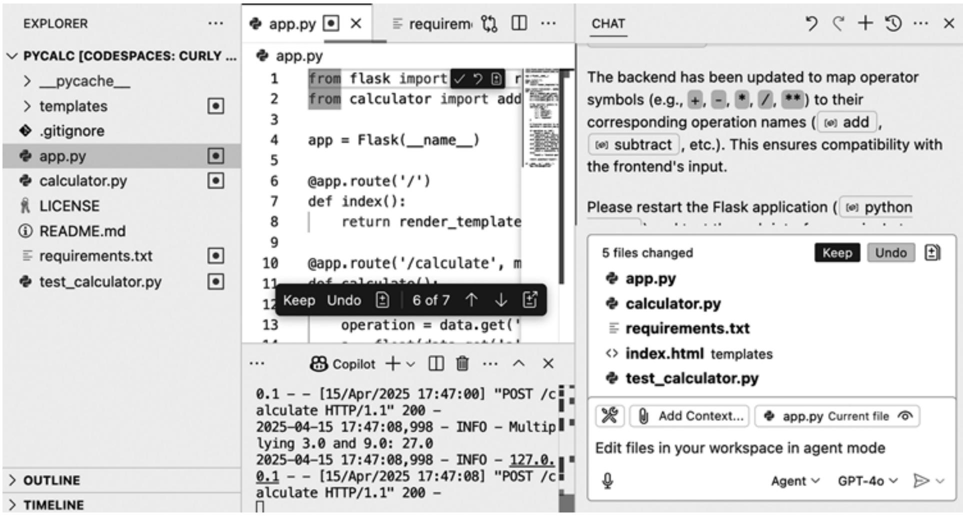
The Flask-based web interface has been added to ◆ calculator.py. Next, I will create the front-end HTML file (`index.html`)

Рис. 4.24. Агент создает веб-интерфейс

В зависимости от промпта и сложности задачи Copilot может остановиться и предложить код для создания нового файла или же может создать файл самостоятельно. Аналогичным образом он может устанавливать зависимости и выполнять команды терминала либо же предлагать их, чтобы необходимые команды выполняли вы. Если вы не видите автономное поведение, которое ожидали, то попробуйте изменить промпт, чтобы заранее разрешить Copilot выполнять дополнительные действия самостоятельно. Например, можно попросить его создать необходимые файлы или обновить требования.

Иногда Copilot должен будет выполнить несколько итераций в режиме агента, чтобы выдать нужный результат. Например, если при запуске сгенерированных изменений кода возникает проблема, то, возможно, вам придется сделать шаг назад, составить для агента промпт о проблеме и попросить внести дополнительные исправления. При первом запуске примеров этой главы результаты отображались в веб-приложении неправильно. Совершив две дополнительные итерации, Copilot обнаружил проблему и исправил ее.

Любая итерация в режиме агента выполняется по одной и той же схеме: составление плана, обновление контента с помощью предложенных изменений непосредственно в коде и выполнение команды. Результат остается неизменным: набор изменений, который можно запустить для тестирования и просмотреть, выбрав вариант «Принять» или «Отклонить». На рис. 4.25 показан пример после того, как агент исправил проблему с некорректным отображением результатов, о которой я упомянул в предыдущем абзаце.



```
from flask import                    r
from calculator import add

app = Flask(__name__)

@app.route('/')
def index():
    return render_template

@app.route('/calculate', m
def calculate():

    operation = data.get('
```

```
0.1 - - [15/Apr/2025 17:47:00] "POST /c
alculate HTTP/1.1" 200 -
2025-04-15 17:47:08,998 - INFO - Multip
lying 3.0 and 9.0: 27.0
2025-04-15 17:47:08,998 - INFO - 127.0.
0.1 - - [15/Apr/2025 17:47:08] "POST /c
alculate HTTP/1.1" 200 -
```

The backend has been updated to map operator symbols (e.g., +, -, *, /, **) to their corresponding operation names (add, subtract, etc.). This ensures compatibility with the frontend's input.

Please restart the Flask application (python

Рис. 4.25. Решение после пары итераций

На рис. 4.26 показан результат запуска веб-интерфейса калькулятора, созданного агентом Copilot. Как видите, он очень похож на исходное изображение, которое получил Copilot, и работает корректно. Обратите внимание: в промпте мы просили добавить функцию для возведения в степень (**), и в веб-интерфейсе, сгенерированном Copilot, появилась соответствующая кнопка (вторая слева в первом ряду), которой не было на исходном изображении.

Copilot Vision наряду с NES и режимом агента действительно могут упростить создание и обновление кода. Но, конечно, вы должны самостоятельно выяснить причину проблемы, если сгенерированный код работает не так, как предполагалось. Далее мы кратко обсудим, как Copilot помогает выполнить отладку.

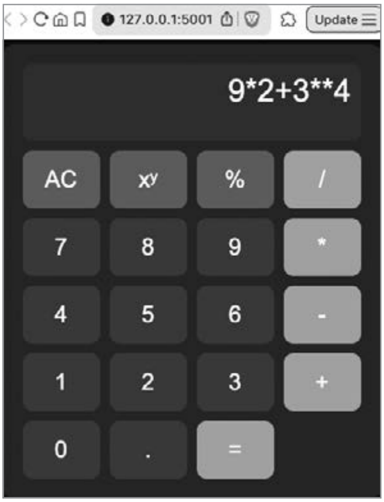
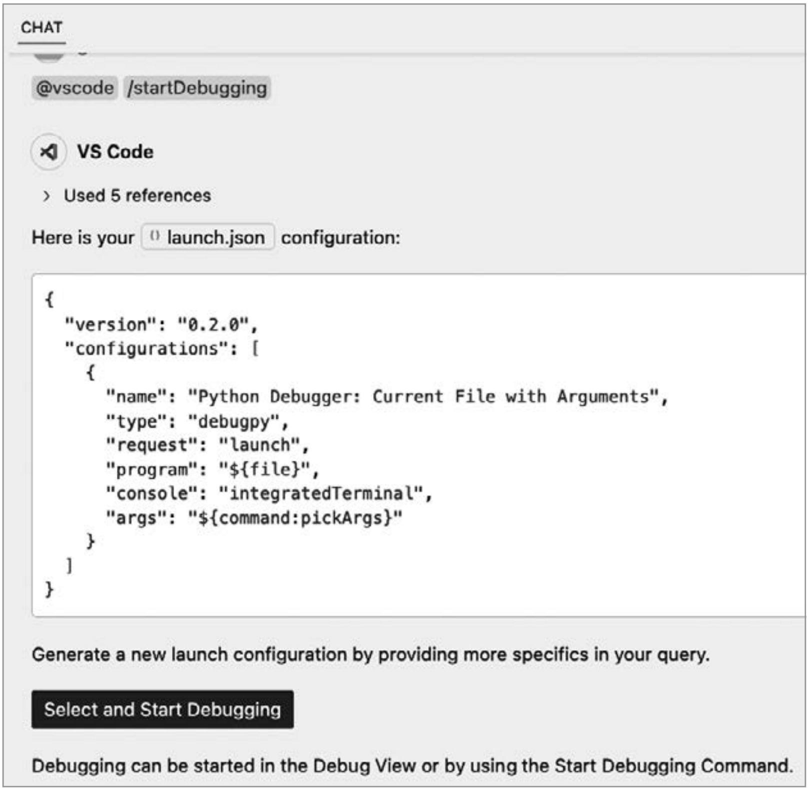


Рис. 4.26. Итоговое веб-приложение

Отладка с использованием Copilot

Ранее мы обсуждали возможности Copilot, которые могут пригодиться при отладке. Мы рассмотрели команду `/explain`, помогающую объяснить код, в котором могут быть проблемы, и команду `/fix`, с помощью которой можно попросить Copilot выдать возможные варианты исправления ошибок. В Copilot есть и другие возможности, которые могут быть полезны при отладке.

В VS Code Copilot может помочь с созданием и настройкой конфигураций откладки. Например, можно использовать слеш-команду `/startDebugging`, чтобы создать файл `launch.json`, если он не существует (рис. 4.27). То же самое можно сделать с помощью промпта в чате.



Here is your launch.json configuration:

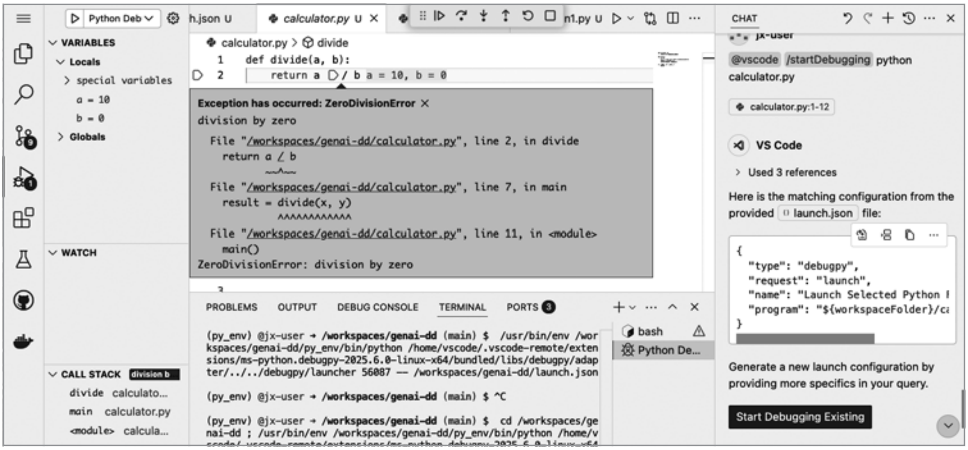
```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Python Debugger: Current File with Arguments",
      "type": "debugpy",
      "request": "launch",
      "program": "${file}",
      "console": "integratedTerminal",
      "args": "${command:pickArgs}"
    }
  ]
}
```

Generate a new launch configuration by providing more specifics in your query.

Debugging can be started in the Debug View or by using the Start Debugging Command.

Рис. 4.27. Использование команды /startDebugging для генерирования конфигурации запуска

Кроме того, с помощью `/startDebugging` можно запустить сеанс отладки. Если при выполнении команды подходящая конфигурация запуска не найдена, то Copilot сначала сгенерирует ее. Затем в том же диалоге он обычно добавляет кнопку, которую можно нажать, чтобы запустить процесс отладки (рис. 4.28).



```python
def divide(a, b):
    return a / b a = 10, b = 0
```

```
Exception has occurred: ZeroDivisionError
division by zero
  File "/workspaces/genai-dd/calculator.py", line 2, in divide
    return a / b
           ~~^~~
  File "/workspaces/genai-dd/calculator.py", line 7, in main
    result = divide(x, y)
             ^^^^^^^^^^^^
  File "/workspaces/genai-dd/calculator.py", line 11, in <module>
    main()
ZeroDivisionError: division by zero
```

```
(py_env) @jx-user → /workspaces/genai-dd (main) $  /usr/bin/env /wor
kspaces/genai-dd/py_env/bin/python /home/vscode/.vscode-remote/exten
sions/ms-python.debugpy-2025.6.0-linux-x64/bundled/libs/debugpy/adap
ter/../../debugpy/launcher 56087 -- /workspaces/genai-dd/launch.json

(py_env) @jx-user → /workspaces/genai-dd (main) $ ^C

(py_env) @jx-user → /workspaces/genai-dd (main) $  cd /workspaces/ge
nai-dd ; /usr/bin/env /workspaces/genai-dd/py_env/bin/python /home/v
```

@vscode /startDebugging python calculator.py

Here is the matching configuration from the provided launch.json file:

```json
{
  "type": "debugpy",
  "request": "launch",
  "name": "Launch Selected Python I
  "program": "${workspaceFolder}/ca
}
```

Generate a new launch configuration by providing more specifics in your query.

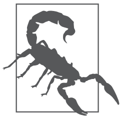
Рис. 4.28. Сеанс отладки с Copilot

В некоторых IDE (например, в Visual Studio) в интеграцию Copilot входит *помощник по исключениям*. Это означает, что если в ходе отладки возникнет исключение (например, `IndexOutOfRangeException`), то появится всплывающее окно, в котором вам будет предложено обратиться за помощью к Copilot. Если вы это сделаете, то он предоставит объяснение ошибки, предложит исправление и может даже применить исправление прямо в IDE. Подробнее об этом — в документации (https://oreil.ly/QopeF).

Если вы проводили юнит-тестирование с помощью Test Explorer в Visual Studio 2022, то могли заметить кнопку Debug with Copilot. Если вы нажмете ее, то в случае сбоя теста Copilot будет выполнять следующие действия:

- генерировать подробный план отладки;
- устанавливать точки останова;
- отслеживать критические переменные;
- руководить процессом отладки.

При этом Copilot анализирует значения переменных в точках останова, предоставляя рекомендации и предложения в режиме обратной связи.



Специфические возможности IDE

Некоторые функции Copilot доступны только в определенных IDE. Помощник по исключениям и команда Debug with Copilot в Test Explorer на момент написания книги были доступны только в Visual Studio (но не в других IDE, таких как VS Code).

Кроме того, Copilot предоставляет в некоторых IDE еще одну команду — `copilot-debug`. Вы можете запустить ее из терминала в своем приложении, чтобы начать сеанс отладки. Например:

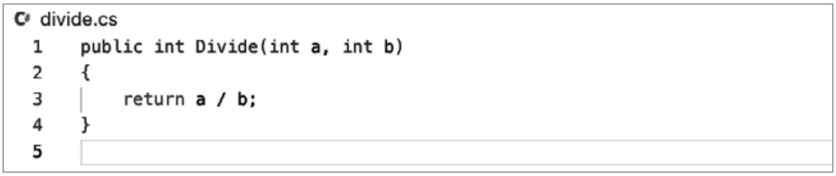
```
copilot-debug node app.js
```

Проблемы, вызванные использованием copilot-debug

Использовать команду copilot-debug может быть удобно в терминале, но в некоторых случаях она приостанавливает работу, ожидая подключения к IDE. Если при ее запуске у вас возникнут проблемы, то попробуйте использовать для запуска сеанса отладки команду /startDebugging или встроенную функциональность IDE.

Наконец, есть еще один прием, который можно использовать для отладки проблем, связанных с функциональностью Copilot Vision, о которой мы говорили в предыдущем разделе, — можно поручить Copilot отладку проблем по снимку экрана.

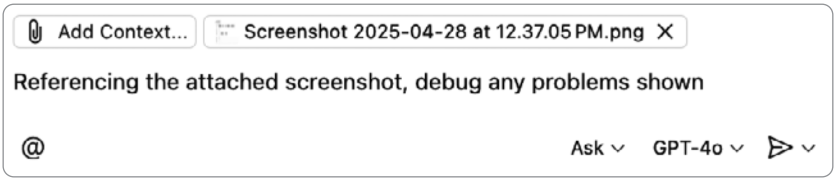
Допустим, у вас есть снимок экрана с простым кодом C# (рис. 4.29), хранящийся в файле `Screenshot 2025-04-28 at 12.37.05 PM.png`.



```
public int Divide(int a, int b)
{
    return a / b;
}
```

Рис. 4.29. Снимок экрана с кодом C#

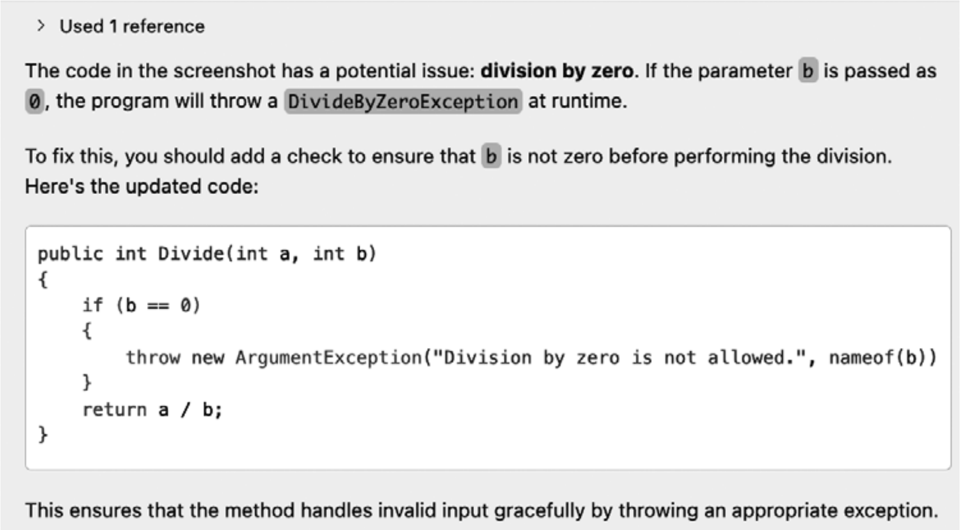
Этот файл можно прикрепить к диалогу в чате, а затем попросить Copilot выполнить отладку любых проблем, связанных с этим файлом. На рис. 4.30 показан один из возможных промптов.



Referencing the attached screenshot, debug any problems shown

Рис. 4.30. Промпт для отладки по снимку экрана

Проанализировав промпт и снимок экрана, Copilot выдает подробную информацию о необработанном исключении, которое может возникнуть из-за деления на ноль, и предлагает код для решения этой проблемы (рис. 4.31).



The code in the screenshot has a potential issue: **division by zero**. If the parameter `b` is passed as `0`, the program will throw a `DivideByZeroException` at runtime.

To fix this, you should add a check to ensure that `b` is not zero before performing the division. Here's the updated code:

```
public int Divide(int a, int b)
{
    if (b == 0)
    {
        throw new ArgumentException("Division by zero is not allowed.", nameof(b))
    }
    return a / b;
}
```

This ensures that the method handles invalid input gracefully by throwing an appropriate exception.

Рис. 4.31. Результаты анализа снимка экрана

В конечном счете независимо от того, используете ли вы встроенные функции отладки в Copilot, чат-интерфейс остается одним из лучших инструментов. Когда вы сталкиваетесь с проблемой и не понимаете ее причину, попробуйте задать Copilot вопрос. Помните, что чем конкретнее запрос (например, «Почему этот вызов API завершается ошибкой при корректных входных данных?»), тем лучше результат.

Резюме

В этой главе мы рассмотрели некоторые продвинутые инструменты для редактирования кода и управления рабочими процессами: NES, Copilot Edits, режим агента и Copilot Vision. Если набор изменений ограничен, то выбрать нужный инструмент может быть непросто. В табл. 4.1 были даны рекомендации, которые помогут понять, когда использовать каждый из них.

Различия можно кратко описать так:

- NES работает на уровне редактирования кода (один файл);
- Copilot Edits функционирует на уровне сеанса проекта (несколько файлов);
- режим агента работает на уровне системы (код + терминал + тестирование).

Для NES предоставляет возможность быстро вносить исправления благодаря встроенным предложениям. Copilot Edits отлично подходит для внесения управляемых изменений в несколько файлов. Режим агента позволяет обрабатывать задачи полного жизненного цикла разработки, такие как обновления кода и его создание, проверка и выполнение.

Контролировать работу описанных функций можно с помощью механизмов Keep/Undo и возможности *отката* изменений.

Copilot Vision позволяет Copilot извлекать информацию из изображений: снимков экрана, на которых показана проблема, схем проектирования, интерфейсов приложений и многого другого. Получив информацию, Copilot может отвечать на вопросы об изображении, проводить анализ, создавать текст или код на основе изображения либо брать части изображения для решения задач или выполнения операций. Использовать Copilot Vision очень просто: изображение просто перетаскивается/вставляется в область ввода в чате (независимо от того, какой режим чата выбран в этот момент).

Стоит отметить еще одну важную деталь, связанную с режимом агента. ИИ-агенты совершенствуются: изменяются их возможности, удобство использования, способности взаимодействия со средой и механизмы приспособления к разным задачам. Можно ожидать, что способности ИИ-агента Copilot продолжат развиваться и будут использоваться в других частях IDE и интерфейсов Copilot. При наличии правильных промптов и инструментов возможности агентов практически безграничны.

Далее, важно помнить, что в Copilot есть инструменты, упрощающие отладку, например команда `/startDebugging`. Вы даже можете попросить Copilot проанализировать снимок экрана и выполнить отладку показанного на нем кода. Наряду с другой функциональностью, позволяющей получать объяснения кода и решать проблемы средствами Copilot, эти инструменты помогут быстрее найти корень проблемы. Но важно заметить, что не все средства отладки Copilot поддерживаются во всех IDE. Наиболее эффективный подход — задавать конкретные вопросы о проблемах напрямую в чат-интерфейсе Copilot.

Итак, вы ознакомились с базовым и расширенным вариантами использования Copilot. Теперь можно перейти к конкретным задачам, которые можно решить с его помощью, — к генерированию тестов, созданию документации и переводу кода на другой язык. Делегируя эти задачи Copilot, вы сможете сэкономить много времени на выполнении рутинных задач и сосредоточиться на более интересных и творческих моментах написания кода. Применение Copilot для решения таких задач мы обсудим в нескольких следующих главах и начнем с того, как использовать его для тестирования.

ГЛАВА 5

Тестирование с использованием Copilot

Вы уже понимаете, как работает Copilot и как взаимодействовать с ним из редактора и чат-интерфейсов. Теперь мы можем обсудить, как с его помощью повысить продуктивность вашей работы. Он упрощает решение рутинных задач, которые могут отнимать много времени и ресурсов. Автоматизируя такую работу, вы сможете сосредоточиться на более сложных задачах, необходимых для создания программного обеспечения.

Эта глава посвящена одной конкретной возможности: использованию Copilot для генерирования тестов. Вы узнаете, как он:

- дает рекомендации по тестированию;
- создает стандартные тестовые сценарии для интеграционного и юнит-тестирования;
- создает тесты для граничных случаев;
- обрабатывает пользовательские инструкции по тестированию;
- пишет тесты, используя выбранный вами фреймворк;
- помогает внедрять лучшие практики, такие как разработка через тестирование (https://oreil.ly/-nm1N);
- использует режим агента Copilot для помощи в создании тестов.

Генеративный ИИ и тестирование

При создании тестов результаты, выданные Copilot, могут сильно различаться по контенту, пригодности и даже точности. Обычно это зависит от объема предоставленного контекста, интерфейса и промпта.

Ввиду вероятностной природы генеративного ИИ нет никаких гарантий, что вы увидите именно то, что хотите. Следовательно, очень важно проверить предложенные тесты и убедиться, что они корректны и подходят для своей цели. Если вы получили не то, чего ожидали, то, возможно, тесты необходимо отредактировать или уточнить промпт и попробовать снова.

Прочитав эту главу, вы сможете использовать Copilot разными способами, чтобы обеспечить необходимое тестовое покрытие.

Для начала зададим Copilot самый общий вопрос о тестировании: «Как мне протестировать свой код?»

Как тестировать

Умение быстро осваивать новые языки программирования и фреймворки — стандартное требование для большинства профессионалов в области программирования. Вероятно, вы уже научились писать тесты на тех языках, которыми пользуетесь регулярно. Однако переход на другую среду разработки или миграция кода могут создать серьезные трудности. Как я уже говорил, на этот случай в Copilot есть полезная функция — вы можете спросить, как написать код или перевести его на другой язык. То же самое применимо и к тестированию.

Перед отправкой запроса держите открытым файл с кодом

Получить лучшие ответы на промпты и запросы, которые мы рассмотрим в этой главе, можно в том случае, если один или несколько файлов вашего проекта открыты в редакторе. Если активных файлов нет, то Copilot может привести более общие объяснения, соответствующие типу вашего проекта, вместо того чтобы давать конкретные рекомендации по содержимому проекта.

В качестве примера мы будем использовать проект для Kubernetes Operator, написанный на Go. Предположим, что вы только осваиваете этот язык и не имеете представления о том, как протестировать только что написанный код. Вы можете поискать информацию на Stack Overflow (https://stackoverlow.com) или других форумах и сайтах либо спросить у коллег. А можете задать вопрос Copilot:

```
Как мне добавить тесты в свой проект?
```

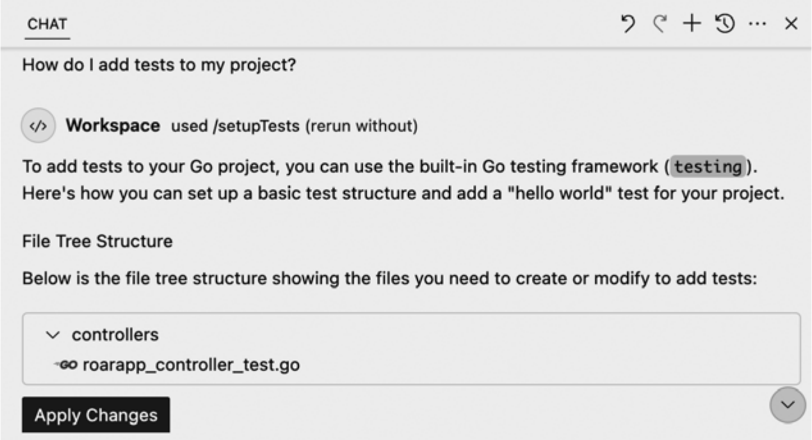
Более конкретные запросы

Если вам нужно более конкретно указать часть проекта, которую вы хотите протестировать, то укажите в промпте имя файла, используйте чат-переменную или добавьте элементы контекста, как объяснялось в главе 3. Например:

```
Как протестировать код в #file:prime.py?
```

Сначала Copilot анализирует проект. Промпт сформулирован без конкретики, поэтому Copilot объясняет, как создать пример теста для вашего проекта. Чтобы предоставить базовую информацию, Copilot использует *участника чата* и *слеш-команду* по умолчанию `@workspace /setupTests`.

Copilot предоставляет сводку своих предложений и кнопки для внесения изменений и выполнения действий (рис. 5.1).



How do I add tests to my project?

Workspace used /setupTests (rerun without)

To add tests to your Go project, you can use the built-in Go testing framework (`testing`). Here's how you can set up a basic test structure and add a "hello world" test for your project.

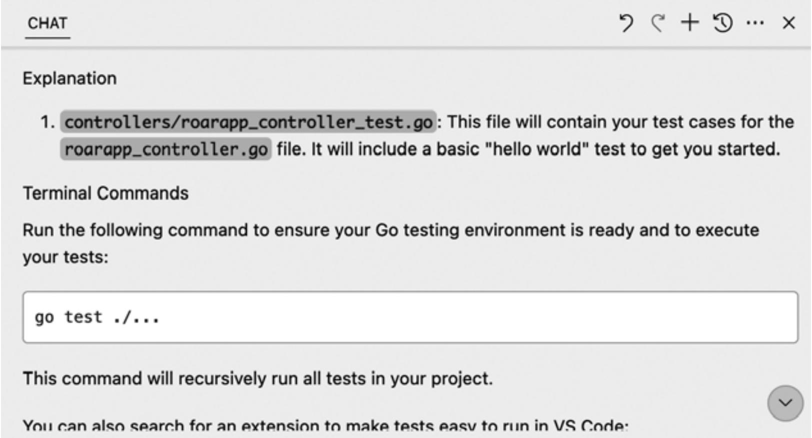
File Tree Structure

Below is the file tree structure showing the files you need to create or modify to add tests:

controllers
roarapp_controller_test.go

Рис. 5.1. Исходные предложения для добавления тестов

В том же ответе Copilot дает объяснение своих предложений и указывает команду терминала для запуска теста (рис. 5.2).



Explanation

1. `controllers/roarapp_controller_test.go`: This file will contain your test cases for the `roarapp_controller.go` file. It will include a basic "hello world" test to get you started.

Terminal Commands

Run the following command to ensure your Go testing environment is ready and to execute your tests:

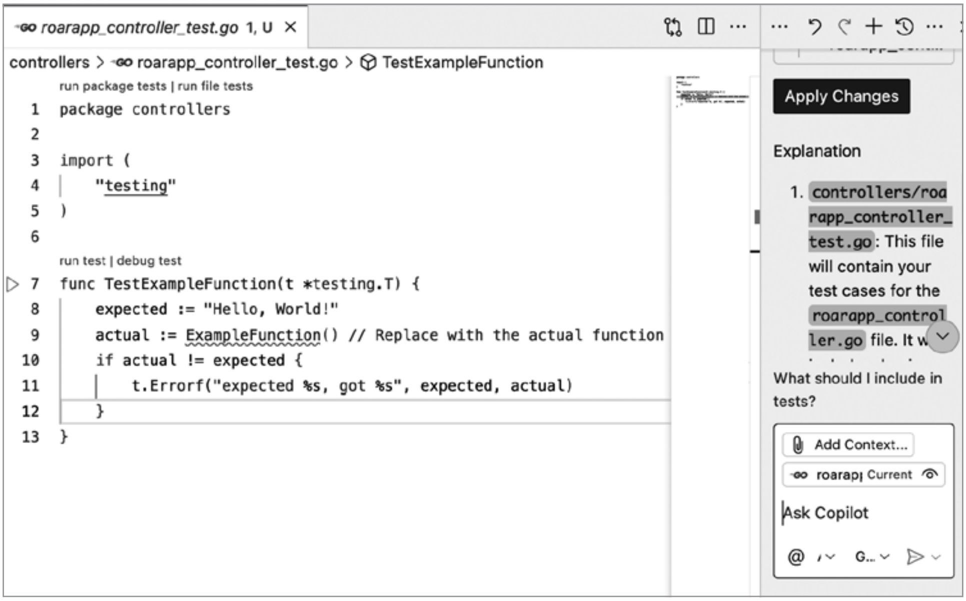
```
go test ./...
```

This command will recursively run all tests in your project.

You can also search for an extension to make tests easy to run in VS Code:

Рис. 5.2. Дальнейшие объяснения по добавлению тестов

Если нажать кнопку Apply Changes (см. рис. 5.1), то Copilot создаст файл за вас (рис. 5.3, *слева*). Обратите внимание: ответ Copilot содержит информацию о том, как выполнить тесты. Неплохо для одного общего запроса!



```
package controllers

import (
    "testing"
)

func TestExampleFunction(t *testing.T) {
    expected := "Hello, World!"
    actual := ExampleFunction() // Replace with the actual function
    if actual != expected {
        t.Errorf("expected %s, got %s", expected, actual)
    }
}
```

1. `controllers/roarapp_controller_test.go`: This file will contain your test cases for the `roarapp_controller.go` file. It w

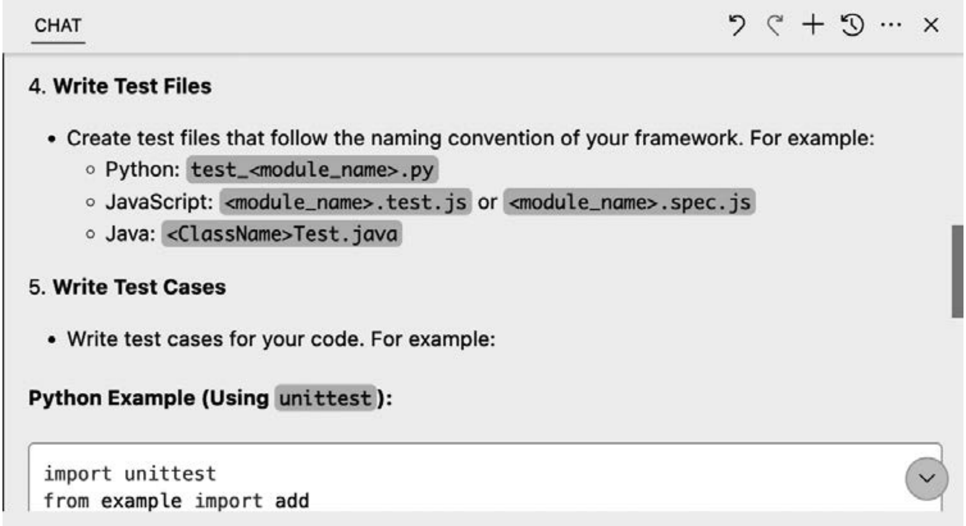
What should I include in tests?

Рис. 5.3. Создание файла нажатием кнопки Apply Changes

Цель команды `/setupTests` — выполнить макроуровневую настройку тестового кода и связанных изменений. На момент написания книги эта функция все еще остается экспериментальной, и ее полезность будет зависеть от конкретной ситуации. Аналогично использование `@workspace` по умолчанию не всегда дает наилучшие результаты. Возможно, вы получите результаты получше, если при первоначальной генерации тестов не станете применять настройки по умолчанию. Если вы используете предыдущий запрос в режиме rerun without, то получите более полный набор действий, которые вам нужно выполнить.

1. Выберите фреймворк тестирования.
2. Подготовьте его.
3. Создайте каталог для тестов.
4. Напишите файлы с тестами.
5. Создайте тестовые случаи.
6. Запустите тесты.
7. Автоматизируйте процесс тестирования.

Вместе с тестами Copilot выдаст более подробное описание того, что нужно сделать, а также примеры кода. На рис. 5.4 приведен фрагмент результатов при использовании режима rerun without.



4. **Write Test Files**

- Create test files that follow the naming convention of your framework. For example:
  - Python: `test_<module_name>.py`
  - JavaScript: `<module_name>.test.js` or `<module_name>.spec.js`
  - Java: `<ClassName>Test.java`

5. **Write Test Cases**

- Write test cases for your code. For example:

Python Example (Using `unittest`):

```
import unittest
from example import add
```

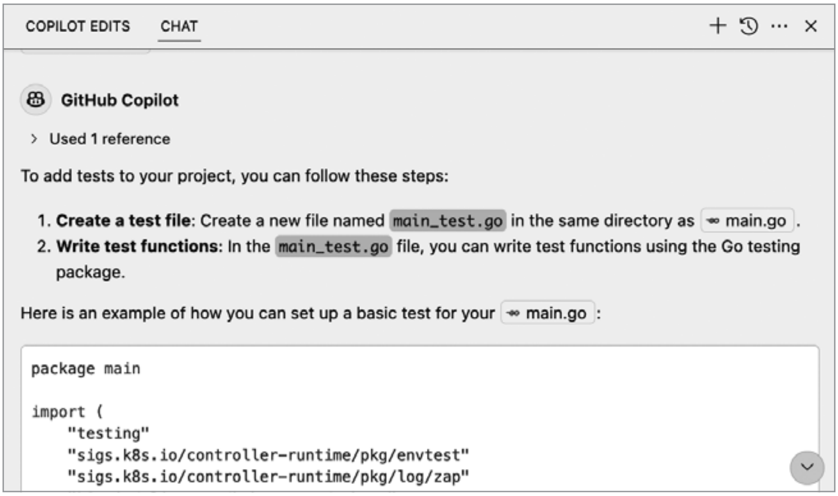
Рис. 5.4. Пример вывода при использовании режима rerun without

Во врезке ниже описываются способы обхода команды `/setupTests`. Те же способы можно использовать для обхода других команд, добавляемых в промпты по умолчанию.

ПРОПУСК УЧАСТНИКОВ ЧАТА И СЛЕШ-КОМАНД ПО УМОЛЧАНИЮ

Вы можете сделать так, чтобы промпты, в которых указаны участники чата и слеш-команды по умолчанию, не выполнялись. Для этого можно использовать несколько способов.

1. Просто удалить их из промпта, куда они будут подставлены автоматически.
2. Выполнить операцию с настройками по умолчанию, а затем выбрать режим rerun without в верхней части вывода чата. Промпт будет выполнен снова, но *без* настроек по умолчанию. (На рис. 5.1 ссылка rerun without находится в конце строки Workspace.) На рис. 5.5 показан пример вывода после использования rerun without.
3. Выбрать другой вариант при отправке текста Copilot. Рядом с кнопкой отправки (кнопка со стрелкой в диалоговом окне) находится стрелка поменьше; с ее помощью можно задать различные способы отправки промпта или запроса. Некоторые из этих настроек (рис. 5.6) мы рассматривали в главе 3. Обычно варианты Send или Send to @workspace позволяют получить более релевантные ответы.
4. Наконец, можно отключить команду `/setupTests` в настройках VS Code. Откройте раздел Settings и найдите в нем setupTests (рис. 5.7).



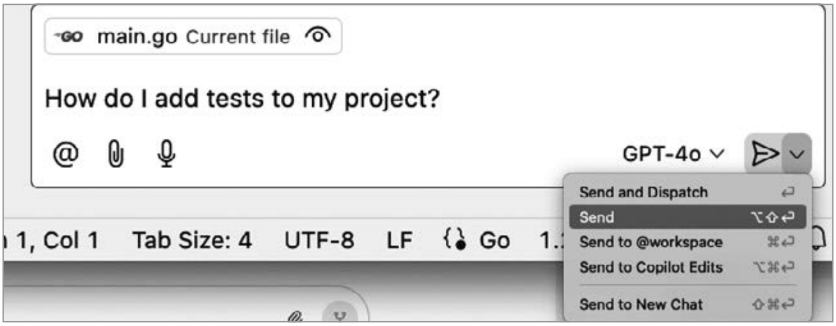
To add tests to your project, you can follow these steps:

1. **Create a test file**: Create a new file named `main_test.go` in the same directory as `main.go`.
2. **Write test functions**: In the `main_test.go` file, you can write test functions using the Go testing package.

Here is an example of how you can set up a basic test for your `main.go`:

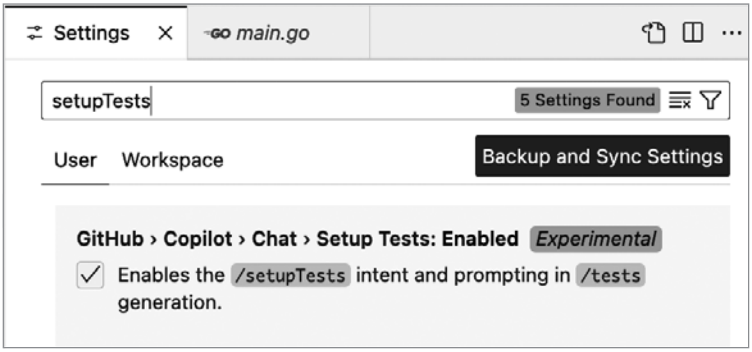
```
package main

import (
    "testing"
    "sigs.k8s.io/controller-runtime/pkg/envtest"
    "sigs.k8s.io/controller-runtime/pkg/log/zap"
```

Рис. 5.5. Вывод после использования ссылки rerun without



How do I add tests to my project?

Рис. 5.6. Параметры передачи промпта



setupTests

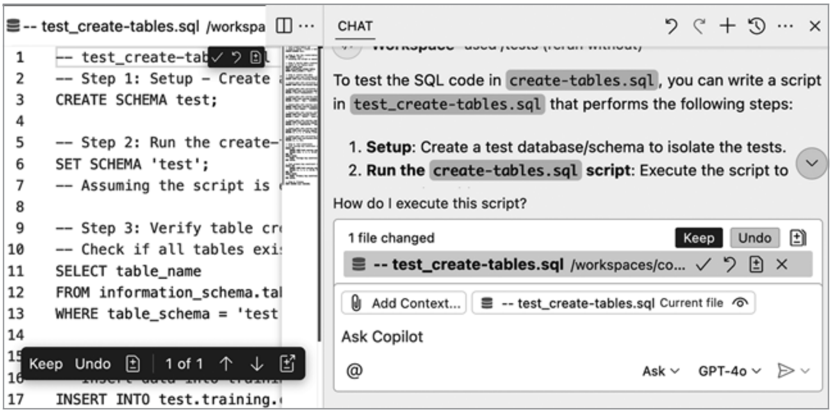
GitHub › Copilot › Chat › Setup Tests: Enabled *Experimental*

☑ Enables the `/setupTests` intent and prompting in `/tests` generation.

Рис. 5.7. Включение/отключение setupTests

В качестве дополнительного примера рассмотрим тему, менее знакомую большинству разработчиков, — тестирование кода SQL. У меня есть большой демонстрационный файл с командами SQL, который я использую в своих учебных курсах. Он создает таблицы для заполнения базы данных об университетских учебных планах, расписании, преподавательском составе и т. д.

Если я открою этот файл в редакторе и спрошу Copilot, как протестировать его содержимое, то Copilot выдаст ответ, показанный на рис. 5.8. Он генерирует пошаговый план тестирования, а затем создает новый файл, реализующий этот план (см. рис. 5.8, *слева*). Предложенный новый файл содержит команды SQL для проведения десяти тестов базы данных.



```
-- test_create-tab
-- Step 1: Setup - Create
CREATE SCHEMA test;

-- Step 2: Run the create-
SET SCHEMA 'test';
-- Assuming the script is

-- Step 3: Verify table cr
-- Check if all tables exi
SELECT table_name
FROM information_schema.tal
WHERE table_schema = 'test

INSERT INTO test.training.
```

To test the SQL code in `create-tables.sql`, you can write a script in `test_create-tables.sql` that performs the following steps:

1. **Setup**: Create a test database/schema to isolate the tests.
2. **Run the `create-tables.sql` script**: Execute the script to

How do I execute this script?

1 file changed

-- test_create-tables.sql /workspaces/co...

Рис. 5.8. Исходный ответ для тестирования SQL

Этот файл можно просмотреть и сохранить как новый набор тестов. Для быстрой генерации этого вывода Copilot использует слеш-команду `/tests`. Но если вам не нужен быстрый вариант, то используйте любую из стратегий rerun without, которые мы обсуждали выше, чтобы Copilot выполнил промпт повторно без слеш-команды.

Мы снова получили результат, который весьма впечатляет, если учесть, что запрос был простым и содержал один файл. Обратите внимание и на то, что описанная схема будет работать с любым языком, а не только с Python.

О доверии к сгенерированным тестам

Мы только начали обсуждать генерацию тестов с помощью Copilot, поэтому я вновь напомню, что их нужно проверять так же, как и любой другой сгенерированный им контент. Тесты, созданные ИИ, могут содержать ложные срабатывания, дублирующиеся утверждения и неправильную логику, основанную на некорректной интерпретации кода. К счастью, код тестов обычно легко читается, что ускоряет его ревью и проверку.

В этом разделе вы узнали, как с помощью Copilot создавать инструкции и код для тестирования в том случае, когда вам нужно начать с нуля. Но чаще вы работаете на языке, который уже знаете, и просто хотите, чтобы Copilot помог создать *шаблонный* код для таких задач, как модульное тестирование. В следующем разделе я покажу несколько способов решения этой задачи.

Создание юнит-тестов

Для таких сценариев, как непрерывная интеграция и разработка через тестирование, нужно писать тесты, и выполнение этой задачи вручную требует больших затрат времени и усилий. С помощью Copilot вы сможете автоматизировать процесс генерирования тестов, используя несколько подходов и выбирая различный уровень сложности.

Во многих случаях тесты, генерируемые Copilot, будут юнит-тестами (еще называют модульными), предназначенными для базового тестирования отдельной функции или процедуры. Но диапазон и глубина тестов, создаваемых Copilot, зависит от промпта и используемого интерфейса. В этом разделе мы рассмотрим разные варианты.

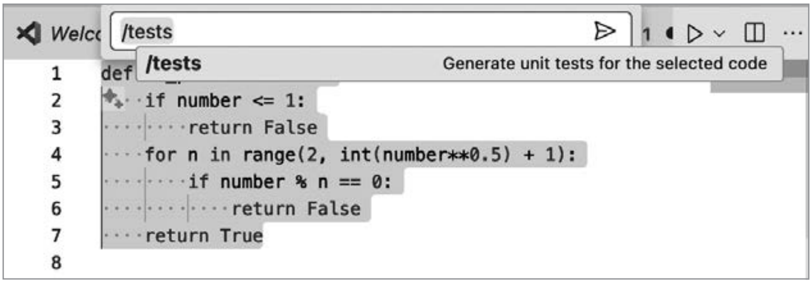
Использование команды /tests

Самый простой способ поручить Copilot создание тестов — использовать встроенную команду `/tests` (в редакторе IDE). Она обрабатывает код, который вы выделили или упоминаете в IDE, а затем пытается создать базовые юнит-тесты, соответствующие коду. Команду можно вводить в любом из чат-интерфейсов, доступных в IDE.

Предположим, что вы снова работаете с простой функцией для определения того, является ли число простым. Реализация в данном случае не имеет значения, одна из версий может выглядеть так:

```
def is_prime(number):
    if number <= 1:
        return False
    for n in range(2, int(number**0.5) + 1):
        if number % n == 0:
            return False
    return True
```

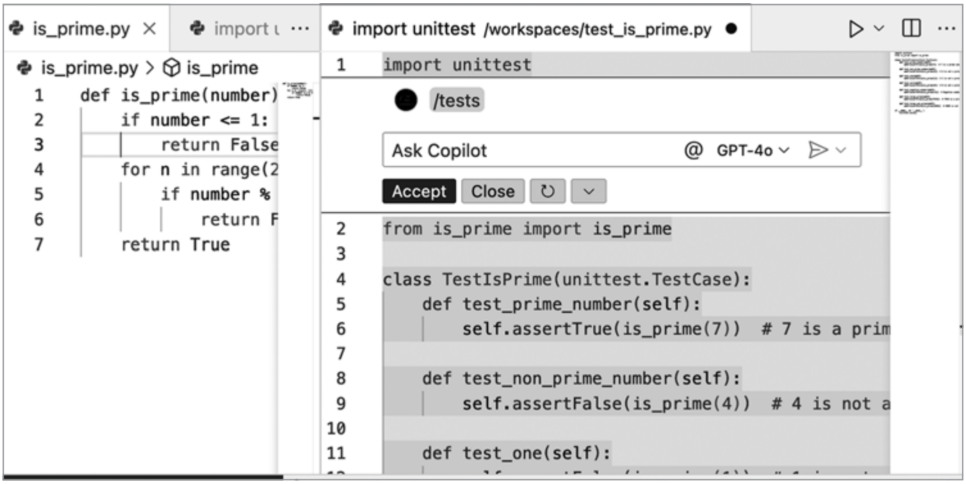
На рис. 5.9 показан пример использования команды `/tests` через встроенный чат. Самый простой способ — выделить код, который нужно протестировать, а затем ввести команду в чат-интерфейсе.



```
def
    if number <= 1:
        return False
    for n in range(2, int(number**0.5) + 1):
        if number % n == 0:
            return False
    return True
```

Рис. 5.9. Вызов Copilot для генерации тестов с помощью команды /tests

Когда команда будет выполнена, Copilot сгенерирует тесты на основе утверждений для простых, составных чисел, единицы, нуля, больших простых и больших составных чисел. Как показано на рис. 5.10, при работе через встроенный чат Copilot предлагает изменения в виде нового файла `test_is_prime.py` в редакторе. Если предложенные тесты вам подходят, нажмите кнопку Accept.



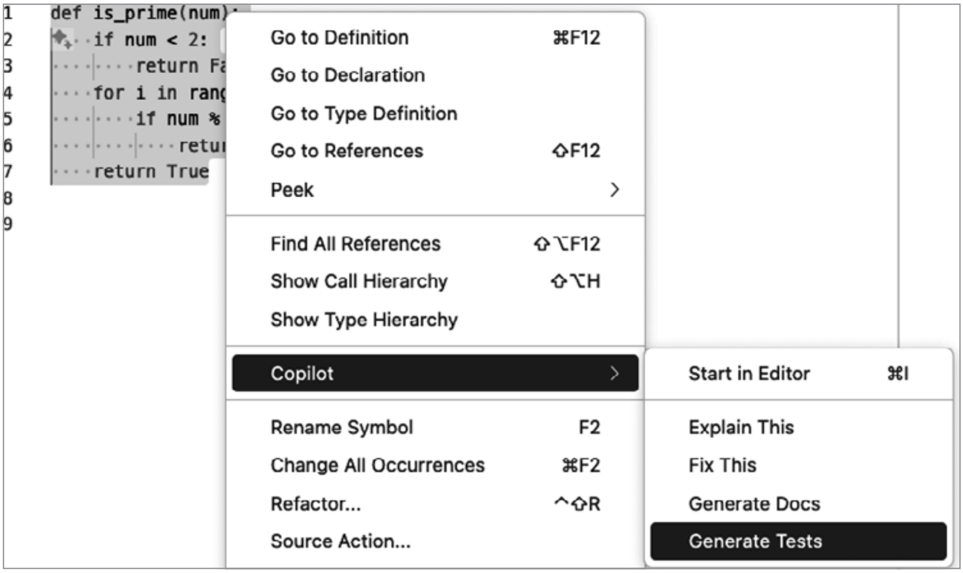
```
def is_prime(number)
    if number <= 1:
        return False
    for n in range(2
        if number %
            return F
    return True
```

```
import unittest
from is_prime import is_prime

class TestIsPrime(unittest.TestCase):
    def test_prime_number(self):
        self.assertTrue(is_prime(7))  # 7 is a prim

    def test_non_prime_number(self):
        self.assertFalse(is_prime(4))  # 4 is not a

    def test_one(self):
```

Рис. 5.10. Тесты из встроенного чата

Такой же результат, как при выполнении команды `/tests` во встроенном чат-интерфейсе, можно получить с помощью команды Copilot ▸ Generate Tests из контекстного меню (рис. 5.11). В большинстве случаев Copilot предпочитает использовать команду `/tests`.

Если вы выполните команду `/tests` в отдельном чат-интерфейсе, то, скорее всего, получите аналогичные результаты, но с более подробными объяснениями в чате о подходе, который Copilot собирается использовать, выбранном фреймворке и типе создаваемых тестов.



```
def is_prime(num):
    if num < 2:
        return Fa
    for i in rang
        if num %
            retu
    return True
```

Рис. 5.11. Генерирование тестов из контекстного меню

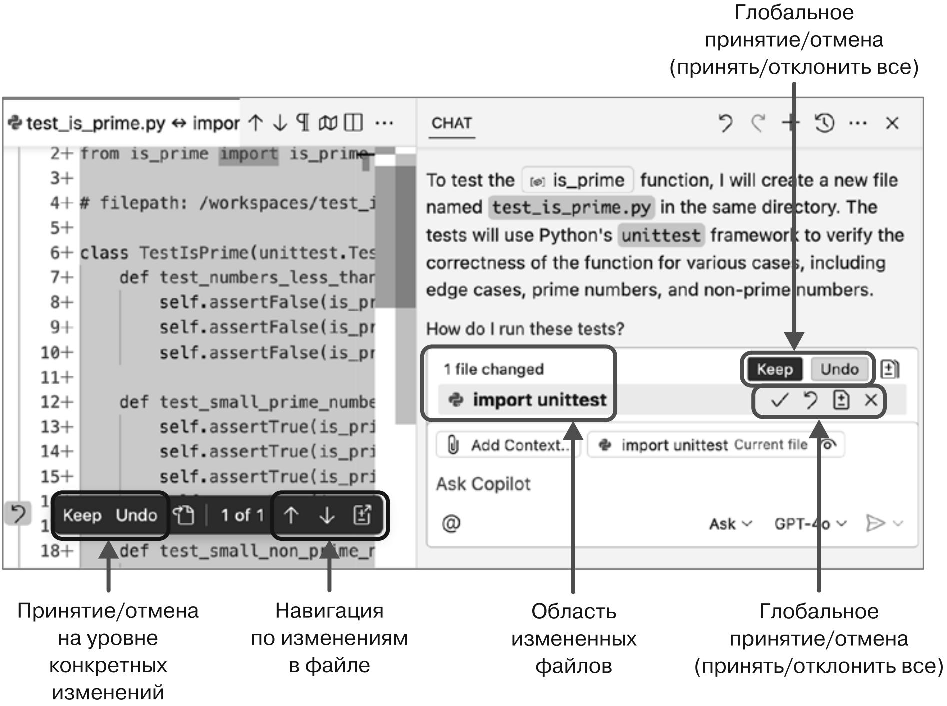
В новом разделе непосредственно над областью ввода текста перечислены все измененные файлы. Для всех файлов доступны кнопки Keep/Undo. Предлагаемые файлы выводятся в виде первой строки, с отдельными элементами управления, позволяющими принять изменения (галочка) и отменить их (стрелка, изогнутая влево). Щелкнув на элементе управления, который выглядит как значок документа с + и –, вы получите сводку различий, возникающих из-за предложенных изменений в редакторе. На рис. 5.12 показана часть вывода.

Изменение элементов и значков

К сожалению, в разных интерфейсах Copilot одни и те же элементы часто изменяются и перемещаются. К тому моменту, когда вы будете читать эту книгу, какие-то из них наверняка изменятся.

Так, для предлагаемых правок действуют следующие универсальные правила:

- будут доступны элементы для принятия/отклонения изменений;
- галочки одновременно выполняют функцию сохранения изменений;
- будет элемент управления для просмотра предлагаемых изменений в виде файла в редакторе или применения этих изменений к этому файлу;
- элементы управления для просмотра различий могут создавать файл с понятным именем, хотя так происходит не всегда. Помните, что вы должны самостоятельно сохранить файлы после того, как просмотрите изменения и примете нужные.



```
from is_prime import is_prime

# filepath: /workspaces/test_i

class TestIsPrime(unittest.Tes
    def test_numbers_less_thar
        self.assertFalse(is_pr
        self.assertFalse(is_pr
        self.assertFalse(is_pr

    def test_small_prime_numbe
        self.assertTrue(is_pri
        self.assertTrue(is_pri
        self.assertTrue(is_pri

    def test_small_non_prime_r
```

To test the is_prime function, I will create a new file named `test_is_prime.py` in the same directory. The tests will use Python's `unittest` framework to verify the correctness of the function for various cases, including edge cases, prime numbers, and non-prime numbers.

How do I run these tests?

Рис. 5.12. Тесты, полученные в результате выполнения команды /tests в главном чате

Эти действия охватывают базовый диапазон сценариев использования. Но что, если нужно сгенерировать дополнительные тесты и/или охватить больше случаев? Этого можно достичь, используя более конкретные промпты. Поговорим о них в следующем подразделе.

Создание тестов с помощью более конкретных промптов

Другой способ создания тестов основан на введении промптов через чат-интерфейс. Использование простого промпта вида «Создай тесты для выделенного кода» обычно приводит к тому, что Copilot повторно выполняет команду `/tests` (если она доступна в вашей IDE). Но вы можете ввести более конкретный промпт, чтобы Copilot предложил тесты для других случаев. Например, если вы уже добавили сгенерированные тесты, то можете выбрать текущий файл с тестами и файл реализации в качестве контекста, а затем спросить у Copilot о других граничных случаях, используя такой промпт:

```
Есть ли другие подобные граничные случаи, которые нужно протестировать?
```

Copilot выдает определение дополнительных тестовых случаев, сопровождая свой ответ пояснениями:

```
Да, есть дополнительные граничные случаи, которые, возможно, стоит протестировать:

1. Наименьшее простое число: убедиться, что функция правильно идентифицирует 2 как
простое число.

2. Четные числа больше 2: проверить четное число (например, 10) и убедиться в том,
что оно не является простым.

3. Очень большое составное число: проверить большое число, которое не является
простым, — например, 10^6.

4. Ввод с плавающей запятой: проверить, как функция справляется с нецелыми входными
значениями (например, 3,5).

5. Строковый ввод: проверить, как функция справляется с недействительными
входными типами (например, "seven").

6. Отрицательное четное число: проверить отрицательное четное число (например, -4),
чтобы убедиться в том, что оно не является простым.
```

Далее Copilot предлагает код, который можно добавить для этих тестовых сценариев:

```
def test_smallest_prime(self):
    self.assertTrue(is_prime(2)) # 2 — наименьшее простое число

def test_even_number_greater_than_two(self):
    self.assertFalse(is_prime(10)); # 10 — это не простое число

def test_very_large_composite_number(self):
    self.assertFalse(is_prime(10**6)) # 10^6 — это не простое число

def test_floating_point_input(self):
    with self.assertRaises(TypeError): # Предполагаем, что это функция
 raises TypeError for non-integers
        is_prime(3.5)

def test_string_input(self):
    with self.assertRaises(TypeError): # Предполагаем, что это функция
 raises TypeError for non-integers
        is_prime("seven")

def test_negative_even_number(self):
    self.assertFalse(is_prime(-4)) # Отрицательные четные числа не являются простыми
```

Возникает естественный вопрос: нельзя ли расширить промпт, чтобы охватить еще больше сценариев? Конечно, можно! Например, можно попросить Copilot добавить тестовые сценарии для других вариантов ввода:

```
Добавь тестовые случаи для других видов ввода
```

Этот промпт приводит к тому, что Copilot добавляет тестовые сценарии, которые вы можете проверить и принять решение:

```
def test_float_input(self):
    with self.assertRaises(TypeError):
        is_prime(7.1)

def test_string_input(self):
    with self.assertRaises(TypeError):
        is_prime("7")
```

Чтобы извлечь максимальную пользу из ответов Copilot, нужно задать ему дополнительные вопросы, выходящие за рамки того, что он может сгенерировать на основе промптов или команд. Помните аналогию с новым участником проекта? Вы можете подтолкнуть новичка к тому, чтобы он вышел за рамки обычных тестовых сценариев и продумал другие возможности. Copilot тоже можно побудить к выдаче более полных результатов.

Если вы предпочитаете создавать быстрые тесты непосредственно в коде, то можете использовать измененный вариант с промптом, описываемый в следующем подразделе.

Создание тестов с помощью комментариев

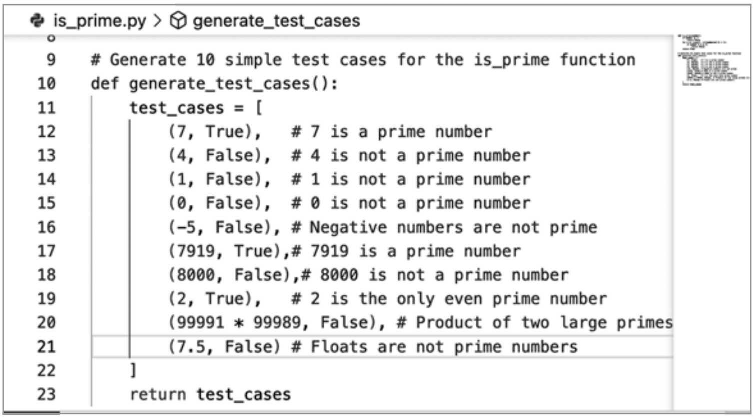
В некоторых ситуациях вам не нужно использовать чат для создания тестов. Представьте, что активно работаете в редакторе и хотите дать Copilot указание наподобие этого: «Вставь сюда несколько простых и быстрых тестов». Это можно сделать, создав для Copilot комментарий-инструкцию. Он предложит отдельные тесты или тестовые функции, и вам останется только принять их. Процесс проходит по той же схеме интерактивного выполнения последовательности операций (принятие, редактирование и т. д.), которая описывалась в главе 2.

Преимущество этого метода заключается в том, что его можно запустить из простой (или сложной) строки комментария в коде. Недостаток — в том, что вам, возможно, придется многократно принимать часть предложений или получать слишком общие предложения, польза которых невелика.

Вот простой пример комментария-инструкции, размещенного непосредственно в коде после функции `is_prime`:

```
# Сгенерируй 10 простых тестовых сценариев для функции is_prime
```

В ответ Copilot предлагает встроенную функцию с набором простых тестовых условий (рис. 5.13). Он автоматически сгенерировал строки 10–23 в ответ на комментарий в строке 9.



is_prime.py > generate_test_cases

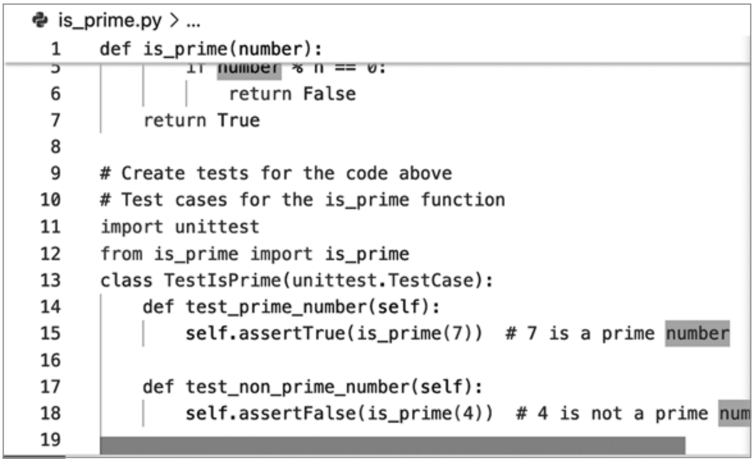
```
 9  # Generate 10 simple test cases for the is_prime function
10  def generate_test_cases():
11      test_cases = [
12          (7, True),   # 7 is a prime number
13          (4, False),  # 4 is not a prime number
14          (1, False),  # 1 is not a prime number
15          (0, False),  # 0 is not a prime number
16          (-5, False), # Negative numbers are not prime
17          (7919, True),# 7919 is a prime number
18          (8000, False),# 8000 is not a prime number
19          (2, True),   # 2 is the only even prime number
20          (99991 * 99989, False), # Product of two large primes
21          (7.5, False) # Floats are not prime numbers
22      ]
23      return test_cases
```

Рис. 5.13. Простой пример тестовых условий, сгенерированных на основании комментария

Можно использовать и более общий комментарий:

```
# Создай тесты для приведенного выше кода
```

В этом случае Copilot генерирует аналогичные тестовые сценарии в других методах, но в том же файле. На рис. 5.14 показан фрагмент сгенерированных тестов. Единственной строкой, введенной вручную, был первый комментарий. Все остальные строки были сгенерированы и приняты на его основе.



is_prime.py > ...

```
def is_prime(number):
        if number % n == 0:
            return False
    return True

# Create tests for the code above
# Test cases for the is_prime function
import unittest
from is_prime import is_prime
class TestIsPrime(unittest.TestCase):
    def test_prime_number(self):
        self.assertTrue(is_prime(7))  # 7 is a prime number

    def test_non_prime_number(self):
        self.assertFalse(is_prime(4))  # 4 is not a prime num

```

Рис. 5.14. Другой пример генерирования тестов на основании комментария

Далее мы рассмотрим еще один способ использования Copilot для тестирования — проверку входных данных функций непосредственно в коде.

Проверка входных данных

Copilot может помочь сгенерировать код для проверки корректности входных данных функции. Возвращаясь к нашему примеру, мы можем дать Copilot следующее указание:

```
Сгенерируй тестовые условия для проверки правильности входных данных функции
```

Copilot реагирует на запрос и предлагает тестовые условия для проверки ввода:

```
Чтобы убедиться в корректности входных данных функции, можно добавить в ее начало
тестовые условия. В данном случае нужно удостовериться, что входное значение
является целым числом и что оно не отрицательно. Это можно сделать так:

def is_prime(num):
    assert isinstance(num, int), "Input must be an integer"
    assert num >= 0, "Input must be a non-negative integer"

    if num < 2:
        return False
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False
    return True
```

Кроме того, можно не указывать конкретно на тестовые условия (`assert`) и просто попросить Copilot сгенерировать проверки:

```
Сгенерируй проверки для контроля правильности входных данных функции
```

Тогда Copilot сгенерирует код для проверки входных данных, но будет использовать более стандартные программные конструкции для выявления любых проблем:

~~~
Чтобы убедиться в том, что функция `is_prime` получила корректные входные данные,
можно добавить в ее начало. Это можно сделать так:

```python
def is_prime(num):
    if not isinstance(num, int):
        raise TypeError("Input must be an integer")
    if num < 0:
        raise ValueError("Input must be a non-negative integer")

    if num < 2:
        return False
~~~

```
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False
    return True
```
```

```
В этом коде команды `if` выдают исключение `TypeErrore, если полученное
значение не является целым числом, или исключение `ValueError, если это
отрицательное число. Таким образом, вы можете быть уверены в том, что
входные данные функции корректны.
```

Каждый из подходов может быть полезен в разных ситуациях в зависимости от того, насколько простые результаты вам нужны, и от требований к охвату тестирования. Помните: вы взаимодействуете с ИИ, поэтому получение конкретных результатов (в отличие от общих или быстро сгенерированных встроенных вариантов) напрямую зависит от формулировки промпта и указания контекста. В зависимости от вашего прогресса работы над проектом и опыта использования языков и фреймворков вы можете обнаружить, что лучше всего сочетать эти подходы.

Прежде чем завершать раздел, посвященный созданию юнит-тестов, приведу общий шаблон промпта для создания набора таких тестов (адаптированный по документации Copilot (https://oreil.ly/NOS8j)):

`Сгенерируй разносторонний набор юнит-тестов для` *`<метод/функция>`* `в` *`<язык/фреймворк>`*`. Напиши тестовые методы, покрывающие широкий диапазон сценариев, в том числе граничные случаи, обработку исключений и проверку данных.`

Замените конструкции в угловых скобках конкретными названиями. Шаблон состоит из следующих ключевых элементов.

Область видимости

В шаблоне явно запрашивается разносторонний набор юнит-тестов, поэтому ИИ не ограничивается базовыми тестами и покрывает разнообразные потенциальные ситуации использования.

Сценарии

Шаблон указывает, что тесты должны включать:

- граничные случаи для проверки критических условий;
- обработку исключений для ошибок, которые будут возникать при выполнении недопустимых действий;
- проверку данных, которая гарантирует, что входные данные могут быть корректно обработаны методами.

Теперь, когда вы знаете, как генерировать юнит-тесты, посмотрим, как использовать Copilot для написания интеграционных тестов.
```

## Создание интеграционных тестов

Итак, с помощью Copilot можно создавать интеграционные тесты. Попросите его сосредоточиться на взаимодействии нескольких компонентов, а не на генерации тестов для изолированных модулей кода. Для этого необходимо написать промпты с четкими инструкциями, в которых подчеркивается совместная работа сервисов, модулей и/или внешних систем.

Вот ключевые моменты, влияющие на успешность создания Copilot интеграционных тестов.

*Создание конкретных промптов*

Попросите Copilot написать интеграционные тесты для проверки взаимодействия между компонентами (а не просто генерировать тесты для кода).

*Использование макетов для внешних зависимостей*

Интеграционным тестам часто приходится имитировать внешние сервисы, такие как базы данных или API. Вы можете попросить Copilot использовать макеты, или моки (mocks), чтобы тесты могли проверить функциональность, не используя реальные реализации.

*Запросы о покрытии взаимодействий*

Попросите Copilot убедиться в том, что методы вызываются, данные между компонентами передаются ожидаемым образом и происходят ожидаемые побочные эффекты.

Общий шаблон промпта для интеграционных тестов может выглядеть примерно так:

```
Напиши интеграционные тесты для <метод/функция> в <класс>. Используй макеты для
моделирования <внешний сервис> и убедись, что они правильно работают с <условие>.
```

(Как и ранее, текст внутри угловых скобок нужно заменить названиями, соответствующими вашему конкретному сценарию использования.) Предположим, что у нас есть приложение Express с двумя модулями.

`UserService`

Обрабатывает пользовательские данные.

`AuthController`

Обрабатывает запросы на вход в систему и делегирует их `UserService`.

Мы хотим написать интеграционные тесты для проверки правильности взаимодействия `AuthController` с `UserService` и уровнем HTTP. Вот пример промпта для решения этой задачи:

```
Напиши интеграционные тесты для конечной точки login в AuthController.
Сгенерируй макет для UserService, чтобы смоделировать проверку данных
пользователя и пароля, и убедись в том, что контроллер возвращает
правильные коды статуса HTTP и сообщения.
```

А это ответ, сгенерированный Copilot:

Пример интеграционных тестов для конечной точки *login* в *AuthController*. Тесты используют Jest и макет сервиса *UserService*, чтобы моделировать поиск пользователя и проверку пароля.

```
const request = require('supertest');
const express = require('express');
const AuthController = require('../controllers/AuthController');
const UserService = require('../services/UserService');

jest.mock('../services/UserService'); // Макетирование UserService

const app = express();
app.use(express.json());
app.use('/auth', AuthController);

describe('AuthController - Login Endpoint', () => {
  afterEach(() => {
    jest.clearAllMocks(); // Очищает макеты после каждого теста
  });

  it('should return 200 and success message for valid credentials',
  async () => {
    // Макетирование UserService.authenticate для возвращения
    // действительного пользователя
    UserService.authenticate.mockResolvedValue({
      id: 1,
      username: 'user1',
      password: 'password1',
    });

    const response = await request(app)
      .post('/auth/login')
      .send({ username: 'user1', password: 'password1' });
    expect(response.status).toBe(200);
    expect(response.body).toEqual({
      message: 'Login successful',
      user: { id: 1, username: 'user1', password: 'password1' },
    });
    expect(UserService.authenticate).toHaveBeenCalledWith('user1',
    'password1');
  });

  it('should return 401 and error message for invalid credentials',
  async () => {
    // Макетировать UserService.authenticate,
    // чтобы вызов возвращал null для недействительных
    // учетных данных
    UserService.authenticate.mockResolvedValue(null);
```

```
    const response = await request(app)
      .post('/auth/login')
      .send({ username: 'user1', password: 'wrongpassword' });

    expect(response.status).toBe(401);
    expect(response.body).toEqual({
      message: 'Invalid credentials',
    });
    expect(UserService.authenticate).toHaveBeenCalledWith('user1',
    'wrongpassword');
  });

  it('should return 400 if username or password is missing',
  async () => {
    const response = await request(app)
      .post('/auth/login')
      .send({ username: 'user1' }); // Пропущен пароль

    expect(response.status).toBe(400);
    expect(response.body).toEqual({
      message: 'Username and password are required',
    });
  });
});
```

```
Ключевые моменты:
1. Макетирование UserService:
   - `jest.mock('../services/UserService')` используется для макетирования
      модуля `UserService`.
   - `UserService.authenticate` макетируется для моделирования разных сценариев
      (действительный пользователь, недействительные учетные данные).

2. Тестовые сценарии:
   - **Действительные учетные данные**: убеждается, что конечная точка
       возвращает статус 200 и сообщение об успехе.
   - **Недействительные учетные данные**: убеждается, что конечная точка
       возвращает статус 401 и сообщение об ошибке.
   - **Отсутствующие поля**: убеждается, что конечная точка возвращает
       статус 400 при отсутствии обязательных полей.

3. Supertest:
   - `supertest` используется для выдачи запросов HTTP к приложению Express
      и проверки ответа.

4. Подготовка:
   - `AuthController` подключен по маршруту `/auth` в тестовом приложении.

Для запуска тестов используется команда:

npx jest
```

Возможно, вам придется выполнить дополнительную настройку кода интеграционных тестов, чтобы они работали так, как задумано, но при этом вы получите надежную отправную точку для дальнейшей работы.

Независимо от того, какие тесты вы создаете с помощью Copilot, возможно, вы захотите адаптировать их под свой проект или под рекомендации по тестированию, которые должны соблюдать. В следующем разделе вы узнаете, как настроить пользовательские инструкции для тестирования с помощью Copilot.

Определение пользовательских инструкций для тестирования

Настроить процесс генерирования тестов для вашего приложения можно с помощью пользовательских инструкций. Они гарантируют, что Copilot сгенерирует тесты, соответствующие выбранным вами фреймворкам, стандартам программирования и рабочим процессам.

Инструкции для генерирования тестов можно настроить на двух уровнях: на пользовательском и на уровне рабочего пространства.

Для конкретного рабочего пространства вы можете добавить инструкции прямо в файл `.vscode/settings.json` вашего проекта. Если сохранить их, то они будут определять поведение Copilot для всех, кто работает с этим проектом.

Чтобы создать инструкции, подходящие для всех проектов, с которыми вы работаете, вы можете добавить их не в файл проекта, а в *глобальные* настройки VS Code.

Кроме того, на любом из этих уровней (пользовательском или уровне рабочего пространства) вы можете создать отдельный файл Markdown с более подробными инструкциями и указать его в настройках. Это полезно при работе со сложными или развивающимися стандартами, которые будут использоваться в нескольких проектах.

При добавлении пользовательских инструкций для тестирования в файл настроек их следует размещать под ключом `github.copilot.chat.testGeneration.instructions`. Вот пример как с конкретными инструкциями, так и со ссылкой на файл:

```
"github.copilot.chat.testGeneration.instructions": [
  {
    "text": "Prefer Mocha and Chai for testing Node.js modules."
  },
  {
    "text": "Use Pytest for all Python test cases."
  },
```

```
  {
    "file": "test-guidelines.md" // Импортирование инструкций из файла
  }
]
```

Содержимое файла `test-guidelines.md` может выглядеть так:

```
Убедиться в том, что все тестовые функции имеют описательные имена.
Смоделировать внешние вызовы API в интеграционных тестах.
```

Возможно, вам придется поэкспериментировать, чтобы найти оптимальную форму инструкций, но после того, как они будут заданы, Copilot будет соблюдать их при выполнении стандартных операций, таких как `@workspace /tests`.

На момент написания книги пользовательские инструкции по тестированию все еще относятся к числу экспериментальных возможностей.

В завершающей части главы я расскажу, как с помощью Copilot использовать фреймворки и создавать тесты до написания кода реализации. Это особенно полезно в таких методологиях, как разработка через тестирование.

Тестирование до написания кода и использование фреймворков

Разработка через тестирование (Test-Driven Development, TTD) уже упоминалась в этой главе. Если вы еще не знаете, что это такое, — речь идет о методологии разработки ПО, в которой тесты для кода пишутся до написания самого кода. TDD считается лучшей практикой или даже обязательным требованием во многих проектах и группах, занимающихся программированием.

Подробнее о TDD

Если вы новичок в TDD или хотите больше узнать об этой практике, то можете поискать информацию в Интернете. Относительно краткое, но информативное руководство можно найти на сайте testdriven.io (https://oreil.ly/VaMiK).

С помощью Copilot вы можете создать тестовые сценарии для TDD, а затем реализовать тестируемый код. Возьмем простой пример с созданием тестового класса и тестов для кода, предназначенного для управления данными о студентах университета. Для тестирования будет использоваться Mockito (https://site.mockito.org).

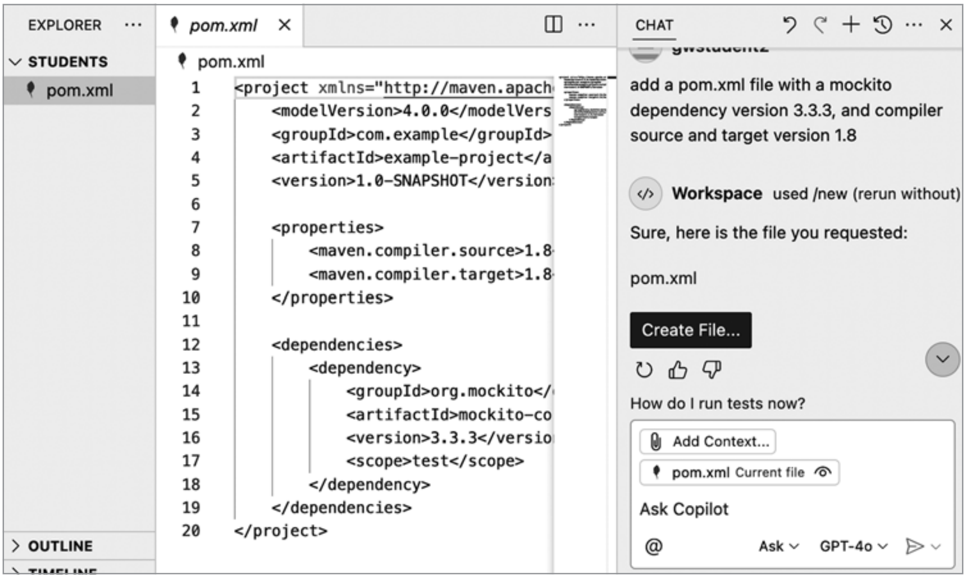
Mockito

Mockito специализируется на управлении зависимостями в приложениях Java и имитации внешних взаимодействий с другими компонентами, не требуя реализации или наличия самих компонентов (то есть он используется для создания макетов). Ранее в этой главе мы показали, как Copilot с помощью макетов может генерировать интеграционные тесты.

Для начала представим, что начинаем работу с пустого проекта `students`, а Copilot создаст файл `pom.xml` (https://oreil.ly/Qx29p) для проекта Java с нужными версиями зависимостей. Вот промпт для этого:

```
Добавь файл pom.xml file с зависимостью mockito версии 3.3.3,
исходной и целевой версией 1.8
```

Как показано на рис. 5.15, с помощью этого промпта Copilot понимает наши намерения и выполняет слеш-команду `/new`, чтобы при необходимости создать структуру. После этого он определяет код файла `pom.xml`, который мы запросили, добавляет ссылку для его просмотра и кнопку для создания этого файла.



```
<project xmlns="http://maven.apach
    <modelVersion>4.0.0</modelVers
    <groupId>com.example</groupId>
    <artifactId>example-project</a
    <version>1.0-SNAPSHOT</version

    <properties>
        <maven.compiler.source>1.8
        <maven.compiler.target>1.8
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.mockito</
            <artifactId>mockito-co
            <version>3.3.3</versio
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

add a pom.xml file with a mockito dependency version 3.3.3, and compiler source and target version 1.8

Workspace used /new (rerun without)

Sure, here is the file you requested:

pom.xml

How do I run tests now?

Ask Copilot

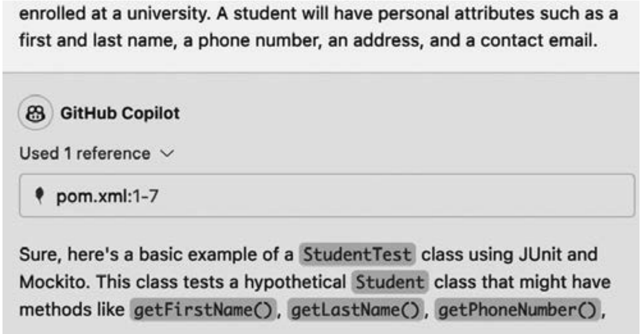
Рис. 5.15. Создание файла pom.xml

Чтобы создать код тестирования, создадим соответствующий тестовый класс и исходный набор тестов. Мы сделаем это в отдельном чат-интерфейсе, так как ожидаем большой объем выходных данных, который лучше разместить в отдельном файле.

УБЕДИТЕСЬ, ЧТО ФАЙЛЫ ДЛЯ КОНТЕКСТА ВЫБРАНЫ

Вероятно, в большинстве случаев явно добавлять ссылку `#file:pom.xml` не понадобится. Если предположить, что файл был сохранен и добавлен в рабочее пространство, то Copilot по умолчанию использует агента `@workspace` и получит контекст Mockito отсюда. Кроме того, созданный им автоматический индекс будет содержать необходимую информацию. Тем не менее в качестве лучшей практики всегда стоит сохранять любые файлы, созданные в редакторе, и добавлять их в рабочее пространство, прежде чем они будут автоматически добавлены в контекст Copilot.

На рис. 5.16 приведен пример использования конкретной ссылки на `pom.xml` для похожего промпта.



enrolled at a university. A student will have personal attributes such as a first and last name, a phone number, an address, and a contact email.

GitHub Copilot

Used 1 reference

pom.xml:1-7

Sure, here's a basic example of a `StudentTest` class using JUnit and Mockito. This class tests a hypothetical `Student` class that might have methods like `getFirstName()`, `getLastName()`, `getPhoneNumber()`,

Рис. 5.16. Использование явной ссылки

В промпте мы просим Copilot обратиться к содержимому файла `pom.xml`, чтобы он знал о том, что необходимо использовать зависимость Mockito:

```
Используя #file:pom.xml, создай класс StudentTest для студентов,
зачисленных в университет, и добавь тесты
```

Copilot возвращает такой ответ:

```
Класс StudentTest с базовыми юнит-тестами, использующими Mockito
для тестирования данных о студентах, зачисленных в университет:

package com.example;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

import java.util.ArrayList;
import java.util.List;
```

~~~
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

class StudentTest {

    private University university;
    private Student student;

    @BeforeEach
    void setUp() {
        university = Mockito.mock(University.class);
        student = new Student("John Doe", "12345");
    }

    @Test
    void testEnrollStudent() {
        // Подготовка
        List<Student> enrolledStudents = new ArrayList<>();
        when(university.getEnrolledStudents()).thenReturn(enrolledStudents);

        // Действие
        university.enrollStudent(student);
        enrolledStudents.add(student);

        // Проверка
        verify(university, times(1)).enrollStudent(student);
        assertEquals(1, university.getEnrolledStudents().size());
        assertEquals(student, university.getEnrolledStudents().get(0));
    }

    @Test
    void testGetStudentDetails() {
        // Подготовка
        when(university.getStudentDetails("12345")).thenReturn(student);

        // Действие
        Student result = university.getStudentDetails("12345");

        // Проверка
        verify(university, times(1)).getStudentDetails("12345");
        assertEquals("John Doe", result.getName());
        assertEquals("12345", result.getId());

    }
}
```

Объяснение:
1. **Зависимости**:
Зависимость `mockito-core` в pom.xml используется для создания макета
для класса `University`
~~~

```
2. **Подготовка теста**:
Метод `@BeforeEach` инициализирует мок-объект `University`
и объект `Student.
3. **Тестовые сценарии**:
   - `testEnrollStudent`: убеждается, что студент поступил в университет,
     и проверяет список зачисленных студентов.
   - `testGetStudentDetails`: убеждается, что по запросу с идентификатором
     возвращается правильная информация о студенте.

Чтобы эти тесты работали, в проекте должны быть реализованы классы `Student`
и `University`. Сообщите мне, если понадобится помощь с их созданием!
```

Copilot выдает полезную информацию и неплохой набор предложенных начальных тестов. Можно взять результат и сохранить его. Но если нам нужен только простой тестовый сценарий для класса `Student`, то добавление таких элементов, как зачисление, может оказаться излишним. Чтобы избежать лишних элементов, изменим структуру промпта, чтобы он более конкретно объяснял, что же нам нужно. Вот один из возможных вариантов:

```
Используя #file:pom.xml, создай только класс StudentTest для студента,
зачисленного в университет. Студент будет иметь такие персональные атрибуты,
как имя и фамилия, телефон, адрес и контактный адрес электронной почты.
```

Промпт содержит слово «только», чтобы Copilot немедленно создал лишь тестируемый класс. В противном случае он может создать и класс реализации. Обновленный код тестирования выглядит так:

```
Класс `StudentTest` для тестирования студента с персональными атрибутами:

package com.example;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class StudentTest {

    private Student student;

    @BeforeEach
    void setUp() {
        student = new Student("John", "Doe", "123-456-7890",
        "123 Main St", "john.doe@example.com");
    }

    @Test
    void testStudentAttributes() {
        // Убедиться, что атрибуты студента заданы правильно
        assertEquals("John", student.getFirstName());
        assertEquals("Doe", student.getLastName());
```

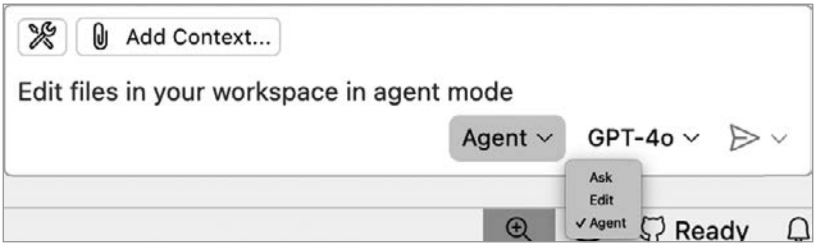
```
        assertEquals("123-456-7890", student.getPhoneNumber());
        assertEquals("123 Main St", student.getAddress());
        assertEquals("john.doe@example.com", student.getEmail());
    }

    @Test
    void testUpdatePhoneNumber() {
        // Обновить телефон и проверить изменения
        student.setPhoneNumber("987-654-3210");
        assertEquals("987-654-3210", student.getPhoneNumber());
    }

    @Test
    void testUpdateAddress() {
        // Обновить адрес и проверить изменения
        student.setAddress("456 Elm St");
        assertEquals("456 Elm St", student.getAddress());
    }

    @Test
    void testUpdateEmail() {
        // Обновить электронную почту и проверить изменения
        student.setEmail("new.email@example.com");
        assertEquals("new.email@example.com", student.getEmail());
    }
}

### Объяснение:
1. **Тестируемые атрибуты**:
   - Имя
   - Фамилия
   - Телефон
   - Адрес
   - Электронная почта
2. **Тестовые сценарии**:
   - `testStudentAttributes`: проверяет, что атрибуты были правильно
     инициализированы.
   - `testUpdatePhoneNumber`, `testUpdateAddress`, `testUpdateEmail`:
     проверяет, что обновления атрибутов работают так, как ожидалось.

Тест предполагает существование класса `Student` с соответствующими
get- и set-методами. Сообщите мне, если понадобится помощь
с созданием этого класса!
```

Новый вариант уже больше похож на отправную точку, которая нам нужна, поэтому мы сохранили его в рабочем пространстве в файле `src/test/java/com/example/StudentTest.java`. Если мы попробуем выполнить тест, то произойдет сбой, поскольку реализации класса `Student` еще нет.

В соответствии с методологией TDD далее нам следует создать минимальный код для прохождения теста. У нас есть Copilot, поэтому мы поручим эту задачу ему, а затем запустим тесты. Мы можем сделать это поэтапно или же использовать режим агента Copilot, как было показано в главе 4. В данном случае мы выберем второй вариант.

Использование режима агента для тестирования рабочих процессов

Чтобы продолжить работу над рабочим процессом TDD, сначала необходимо сменить режим с помощью чат-интерфейса (рис. 5.17).



Edit files in your workspace in agent mode

Рис. 5.17. Переключение в режим агента

В режиме агента мы вводим промпт с описанием того, что для нас должен сделать Copilot:

```
Установи все необходимые зависимости для тестирования с mvn, затем запусти тесты
```

Это не обязательно, но если код открыт в редакторе, то можно попросить Copilot использовать его в качестве ключевой ссылки:

```
Используя #editor, создай класс для представления студента.
```

Copilot предлагает команду `brew install maven` для установки Maven и выполняет ее в терминале, когда мы просим его продолжить. Далее он должен перейти к выполнению `mvn test`. Copilot сталкивается с некоторыми ошибками, но определяет, что необходимо для их устранения.

```
mvn test

Компиляция тестов Maven завершилась неудачно из-за следующих проблем:

1. Отсутствие зависимостей JUnit:
Пакет org.junit.jupiter.api не обнаружен, что указывает на отсутствие
зависимости JUnit 5 (Jupiter) в pom.xml.

2. Класс Student не найден: класс Student не опознан,
 что может быть обусловлено неправильной структурой пакета
 или пропущенной компиляцией.
```

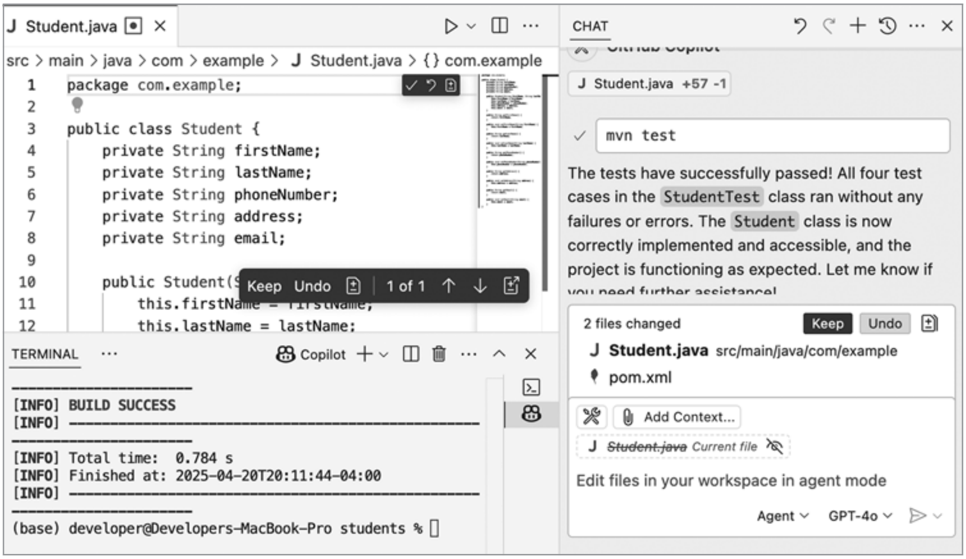
```
3. Аннотации не опознаны: такие аннотации, как @BeforeEach и @Test,
 не опознаны, что также может объясняться отсутствием зависимости JUnit.

Сейчас я:

Добавлю необходимую зависимость JUnit 5 в pom.xml.
Проверю, что класс Student правильно компилируется и доступен для использования.
Снова выполню тесты после исправления этих проблем.
```

Обратите внимание: агент Copilot уже запланировал свои дальнейшие действия для исправления проблем (в разделе «Сейчас я»). После того как вы попросите Copilot продолжать, он сначала обновит файл `pom.xml`, добавляя зависимость JUnit, а затем попытается снова запустить тесты. Они не проходят, поскольку реализация `Student` еще не написана. Вывод Copilot для этого запуска выглядит так:

```
<dependencies>
    <!-- ...существующие зависимости... -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>


Изменения внесены.

Запуск теста Maven завершился неудачей, поскольку класс `Student` не был найден
во время выполнения теста. Это указывает на то, что класс `Student` не был
скомпилирован или не находится в правильной структуре пакетов.

Я проверю местоположение класса `Student, а также то, что он правильно
компилируется и доступен для использования. Затем я снова выполню тесты.
```

И снова агент определил, что нужно делать дальше. После того как вы просмотрите результаты и попросите Copilot продолжать, он сгенерирует новый файл `Student.java` с верным содержимым в правильном месте, а затем снова запустит тесты. На этот раз они завершатся нормально, поскольку агент внес все необходимые изменения. На рис. 5.18 на успешность выполнения указывают следующие элементы:

- новый файл `Student.java` с новым содержимым, ожидающий подтверждения (в левом верхнем углу);
- сообщение об успешной сборке/запуске тестов (внизу слева);
- сообщение агента о том, что все тесты прошли, класс `Student` реализован, а проект работает так, как ожидалось (в правом верхнем углу);
- в области files changed (внизу справа) показано, что агент внес изменения в файлы `pom.xml` и `Student.java`, чтобы все работало.



```
package com.example;

public class Student {
    private String firstName;
    private String lastName;
    private String phoneNumber;
    private String address;
    private String email;

    public Student(S
        this.firstNa
        this.lastName = lastName;
```

```
mvn test
```

```
------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
------------------------
[INFO] Total time:  0.784 s
[INFO] Finished at: 2025-04-20T20:11:44-04:00
[INFO] ------------------------------------------------------------------------
------------------------
(base) developer@Developers-MacBook-Pro students %
```

The tests have successfully passed! All four test cases in the `StudentTest` class ran without any failures or errors. The `Student` class is now correctly implemented and accessible, and the project is functioning as expected. Let me know if you need further assistance!

Рис. 5.18. Успешное завершение с использованием агента

Конечно, то же можно было сделать без агента, выполнив несколько проходов и отдельных действий. Но благодаря использованию агентной функциональности Copilot мы сэкономили немало времени и усилий.

В этой точке процесса можно просмотреть изменения, сохранить обновленные файлы в рабочем пространстве и при необходимости добавить другую функциональность. Аналогичные многоэтапные варианты тестирования рабочих процессов можно реализовать с помощью этого пошагового подхода независимо от того, используете ли вы режим агента.

Как обычно, чем больше контекста и указаний вы дадите Copilot через промпт, файлы рабочего пространства, редактор и т. д., тем выше вероятность, что он предложит качественный контент, которые вы сможете использовать. Но если результат далек от ожидаемого, то не бойтесь уточнять свой промпт/запрос до тех пор, пока не достигнете желаемой цели.

Резюме

В этой главе мы говорили о том, как применять предложения Copilot и функциональность чата в контексте создания инструментов для кода. Вы узнали, что можете использовать разные методы, чтобы Copilot генерировал тесты, соответствующие коду в вашем рабочем пространстве. При этом можно использовать как механизмы встроенного завершения и предложений от Copilot, так и функциональность чата.

В большинстве вариантов генерирования тестов запрос так или иначе передается в чат-интерфейс — либо с помощью слеш-команды, либо через ввод промпта во встроенном или отдельном чат-интерфейсе. Используя функциональность чата для создания любых серьезных или сложных длинных тестов, вы с большой долей вероятности получите наилучший результат. Кроме того, для создания быстрых шаблонных юнит-тестов или простых функций юнит-тестирования можно использовать специальные встроенные комментарии.

Многие базовые команды для создания тестов с помощью Copilot создают юнит-тесты. Кроме того, Copilot помогает создавать интеграционные тесты; для этого формулируйте промпт так, чтобы он охватывал взаимодействие компонентов, использование моков и проверку ожидаемых результатов.

Copilot можно использовать для поддержки лучших практик, таких как TDD. При таком подходе Copilot может генерировать начальный тестовый код, а затем создавать соответствующую реализацию для прохождения тестов. Как и в случае применения всех других подходов, упоминавшихся в этой главе, важно проверить результаты и убедиться, что они пригодны для использования и соответствуют вашим ожиданиям. Вы всегда можете отказаться от предложений или кода Copilot и/или переформулировать промпты и запросы, чтобы получить более точные результаты. Кроме того, вы можете определить более конкретный контекст, который Copilot будет учитывать при генерировании результатов.

Как и код реализации, код тестирования в идеале должен содержать подробные комментарии. Они делают код более читабельным и понятным. Но объяснения и документирование тестовых сценариев или реализаций — это задача, которую разработчики склонны откладывать на потом, чтобы не отвлекаться от работы, а алгоритмы могут измениться. В главе 6 вы увидите, как с помощью Copilot решать эту задачу, практически не прилагая усилий.

ГЛАВА 6

Использование Copilot для документирования и получения объяснений по коду

Все разработчики знают, как важно иметь хорошую документацию для кода. Документация делает кодовую базу более понятной, удобной для проверки и упрощает ее сопровождение в долгосрочной перспективе. Документирование кода — один из важных факторов общего качества кода и созданного на его основе продукта.

Но мы также знаем, что работа по созданию хорошей документации может казаться однообразной и часто воспринимается как задача намного менее приоритетная, чем написание самого кода. Продолжая тему того, как Copilot может помочь разработчику, посмотрим, как он может автоматизировать создание документации.

В этой главе описаны некоторые возможности Copilot по работе с документацией, такие как генерирование документации из редактора и чат-интерфейса. Кроме того, вы увидите, как с помощью Copilot создавать контент для сценариев, которые вы, возможно, не рассматривали, — например, внешней документации для API, функциональной документации и многого другого.

Помимо этого, мы обсудим функциональность Copilot по выдаче объяснений к коду. О ней часто забывают, но она может помочь инженерам всех уровней понять код и алгоритмы. Среди полезных вариантов использования — объяснение кода на незнакомом языке программирования или для незнакомого фреймворка, проверка логики кода с выдачей объяснений и использование Copilot для объяснения команд, выполняемых в терминале.

Сначала рассмотрим, как возможности документирования в Copilot можно использовать на протяжении всего цикла разработки.

Документирование контента

Многие разработчики знают, что могут использовать Copilot для создания документации в заголовках и комментариев в теле кода. Но конкретные механизмы выполнения этих действий не всегда понятны. Можно использовать вспомогательные функции, но их возможности ограничены. Далее мы поговорим о том, как генерировать документацию программы непосредственно в коде и в чат-интерфейсе Copilot.

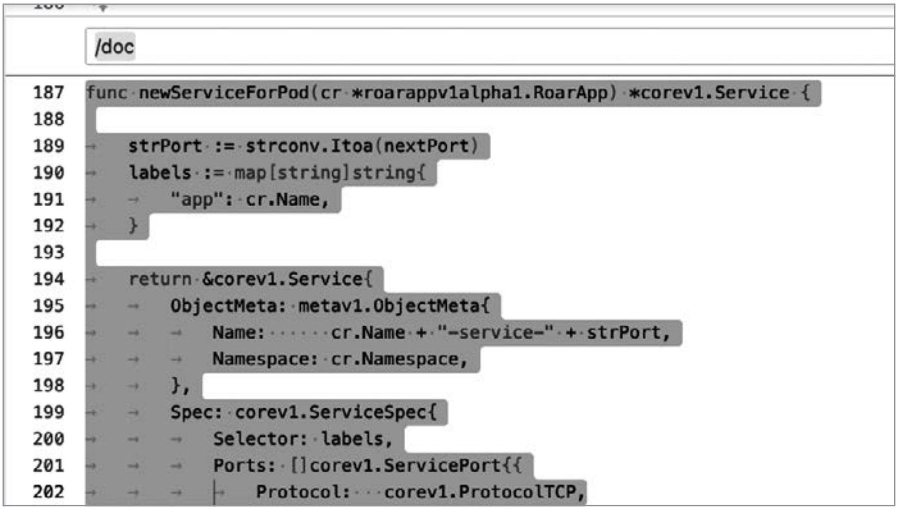
Помимо документации к коду (базовая функциональность), Copilot может создавать и другие виды документации. Он может генерировать документацию для конкретного фреймворка (например, Swagger), документацию API, предназначенную для разработчика, и даже функциональную документацию. Это расширенные сценарии, в которых можно использовать Copilot для создания разных видов документации, и мы рассмотрим их позже. А пока начнем с основ.

Генерирование документации непосредственно в коде

Ниже приведен листинг функции (написанной на Go), которая создает новый сервис Kubernetes для пользовательского ресурса:

```
func newServiceForPod(cr *roarappv1alpha1.RoarApp) *corev1.Service {

    strPort := strconv.Itoa(nextPort)
    labels := map[string]string{
        "app": cr.Name,
    }

    return &corev1.Service{
        ObjectMeta: metav1.ObjectMeta{
            Name:      cr.Name + "-service-" + strPort,
            Namespace: cr.Namespace,
        },
        Spec: corev1.ServiceSpec{
            Selector: labels,
            Ports: []corev1.ServicePort{{
                Protocol:   corev1.ProtocolTCP,
                Port:       8089,
                TargetPort: intstr.FromInt(8080),
                NodePort:   int32(nextPort),
            }},
            Type: corev1.ServiceTypeNodePort,
        },
    }
}
```

Прямой и быстрый способ создать документацию для этой функции непосредственно в коде — вызвать встроенный чат-интерфейс (нажатием Meta+I) и использовать команду `/doc` (рис. 6.1).



```
/doc
```

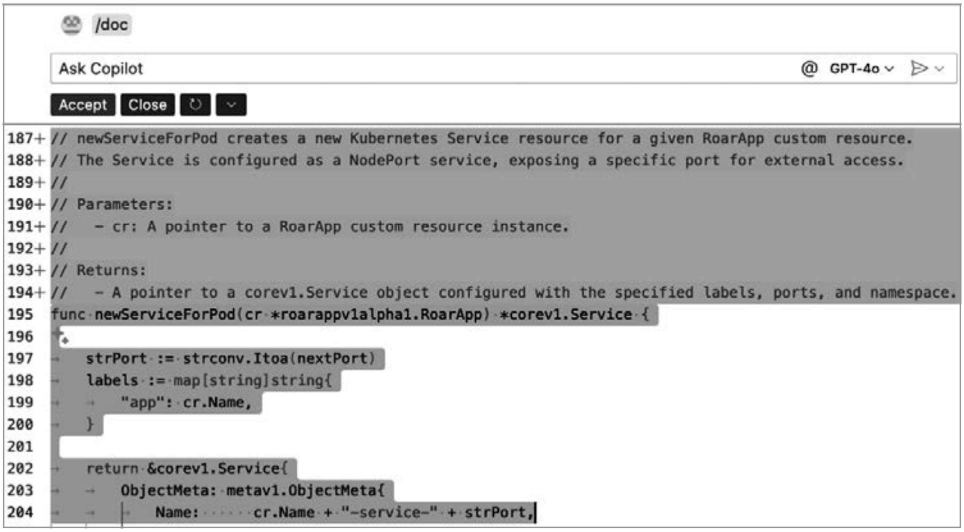
```
func newServiceForPod(cr *roarappv1alpha1.RoarApp) *corev1.Service {

    strPort := strconv.Itoa(nextPort)
    labels := map[string]string{
        "app": cr.Name,
    }

    return &corev1.Service{
        ObjectMeta: metav1.ObjectMeta{
            Name:      cr.Name + "-service-" + strPort,
            Namespace: cr.Namespace,
        },
        Spec: corev1.ServiceSpec{
            Selector: labels,
            Ports: []corev1.ServicePort{{
                Protocol:   corev1.ProtocolTCP,
```

Рис. 6.1. Использование команды /doc

Copilot генерирует базовую документацию для функции. Скорее всего, она будет состоять только из комментариев над функцией, как показано в следующем листинге и на рис. 6.2:

```
// newServiceForPod создает новый ресурс Kubernetes Service
// для заданного ресурса RoarApp.
// Service настраивается как сервис NodePort, предоставляющий
// порт для внешнего доступа.
//
// Параметры:
//   – cr: указатель на экземпляр пользовательского ресурса RoarApp.
//
// Возвращает:
//   – указатель на объект corev1.Service, настроенный с заданными метками,
//   портами и пространством имен.
```

Эта возможность полезна даже в таком виде — добавляются комментарии о назначении функции, ее параметрах и возвращаемых значениях. Но в *теле* функции не хватает дополнительных комментариев или документации. Даже если мы дадим более развернутую команду, например «добавь документацию в мой код», Copilot по умолчанию снова использует команду `/doc` и мы получим аналогичный результат. На момент написания книги это происходит независимо от того, какой чат-интерфейс используется.

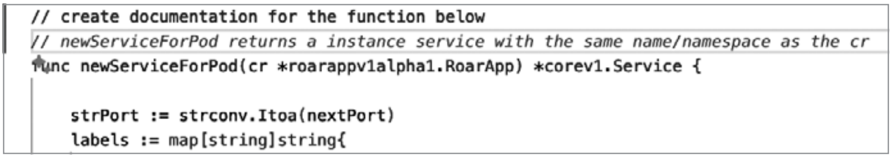


```
// newServiceForPod creates a new Kubernetes Service resource for a given RoarApp custom resource.
// The Service is configured as a NodePort service, exposing a specific port for external access.
//
// Parameters:
//   - cr: A pointer to a RoarApp custom resource instance.
//
// Returns:
//   - A pointer to a corev1.Service object configured with the specified labels, ports, and namespace.
func newServiceForPod(cr *roarappv1alpha1.RoarApp) *corev1.Service {

    strPort := strconv.Itoa(nextPort)
    labels := map[string]string{
        "app": cr.Name,
    }

    return &corev1.Service{
        ObjectMeta: metav1.ObjectMeta{
            Name:      cr.Name + "-service-" + strPort,
```

Рис. 6.2. Сгенерированная документация

СОЗДАНИЕ ДОКУМЕНТАЦИИ ПО КОММЕНТАРИЯМ

Copilot позволяет сгенерировать документацию по комментариям (рис. 6.3).



```
// create documentation for the function below
// newServiceForPod returns a instance service with the same name/namespace as the cr
func newServiceForPod(cr *roarappv1alpha1.RoarApp) *corev1.Service {

    strPort := strconv.Itoa(nextPort)
    labels := map[string]string{
```

Рис. 6.3. Создание документации по комментариям

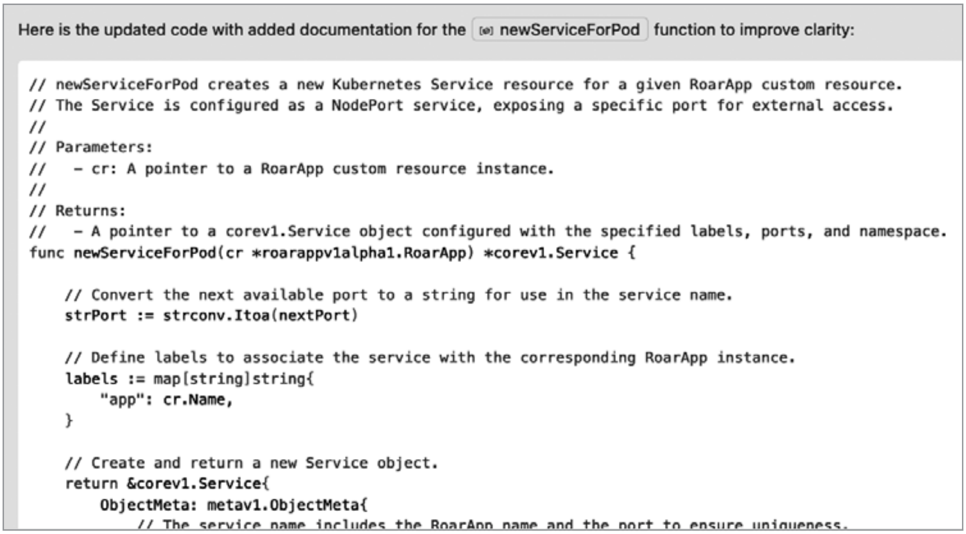
Для активации Copilot необходимо использовать формулировки наподобие «Создай документацию». Но, как видите, результаты получаются довольно упрощенными. И вам придется самостоятельно написать комментарии, чтобы дать Copilot указание создать комментарии.

Помните, что при работе непосредственно в редакторе Copilot выдает короткие, быстрые ответы, которые не нарушают рабочий процесс и которые можно легко просматривать и принимать или отклонять.

Как правило, для получения от Copilot более содержательных ответов можно использовать отдельный чат-интерфейс. Это касается и создания документации, как будет показано далее.

Создание документации в чате

Если вы переключитесь на отдельный чат-интерфейс и дадите Copilot те же указания по добавлению документации в код, то получите более подробный ответ (рис. 6.4).



Here is the updated code with added documentation for the `newServiceForPod` function to improve clarity:

```
// newServiceForPod creates a new Kubernetes Service resource for a given RoarApp custom resource.
// The Service is configured as a NodePort service, exposing a specific port for external access.
//
// Parameters:
//   - cr: A pointer to a RoarApp custom resource instance.
//
// Returns:
//   - A pointer to a corev1.Service object configured with the specified labels, ports, and namespace.
func newServiceForPod(cr *roarappv1alpha1.RoarApp) *corev1.Service {

    // Convert the next available port to a string for use in the service name.
    strPort := strconv.Itoa(nextPort)

    // Define labels to associate the service with the corresponding RoarApp instance.
    labels := map[string]string{
        "app": cr.Name,
    }

    // Create and return a new Service object.
    return &corev1.Service{
        ObjectMeta: metav1.ObjectMeta{
            // The service name includes the RoarApp name and the port to ensure uniqueness.
```

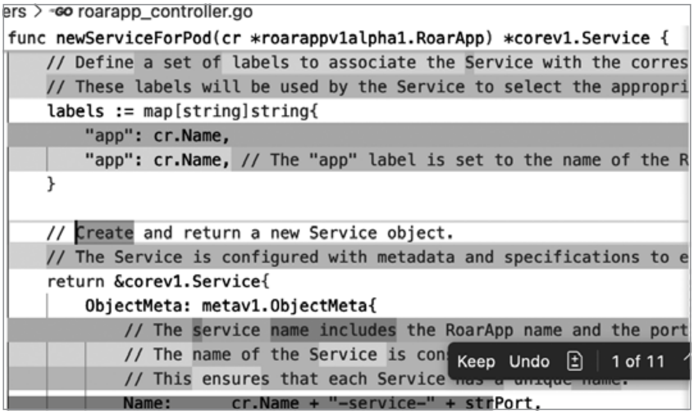
Рис. 6.4. Добавление документации в код из главного чата

Полный листинг вместе со сгенерированными комментариями доступен в репозитории книги на GitHub (https://oreil.ly/zHnkm). Фрагменты показаны ниже. Как видите, Copilot добавил не только существенно больше заголовочных комментариев, но и множество комментариев в тело функции.

```
// newServiceForPod создает новый ресурс Kubernetes Service для заданного
// ресурса RoarApp.
// Service настраивается как сервис NodePort, предоставляющий
// порт для внешнего доступа.
//
// Параметры:
//   - cr: указатель на экземпляр пользовательского ресурса RoarApp.
//
// Возвращает:
//   - указатель на объект corev1.Service, настроенный с заданными метками,
//   портами и пространством имен.
func newServiceForPod(cr *roarappv1alpha1.RoarApp) *corev1.Service {

    // Преобразование следующего доступного порта в строку
    // для использования в имени сервиса
    strPort := strconv.Itoa(nextPort)
```

```
    // Определение метки для связывания сервиса с соответствующим
    // экземпляром  RoarApp.
    labels := map[string]string{
        "app": cr.Name,
    }

    // Создание и возвращение нового объекта Service.
 ...
```

Сделаем еще один шаг. Copilot Chat понимает естественный язык, поэтому можно запросить создание документации, которая будет полезной для каждого, кто будет читать код. Можно попросить Copilot: «Подробно прокомментируй выбранную функцию, чтобы было понятно, что она делает». Результат изменений по сравнению с исходной (непрокомментированной) функцией можно найти в репозитории GitHub (https://oreil.ly/jTDZV). Если не считать некоторых изменений в формулировках, то главное различие заключается в дополнительных строках комментариев для каждого главного раздела, содержащих описание того, как в программе используются отдельные составляющие. Кроме того, комментарии добавляются в конце некоторых строк с идентификаторами, которые стоит пояснить. Фрагмент этих изменений показан на рис. 6.5.

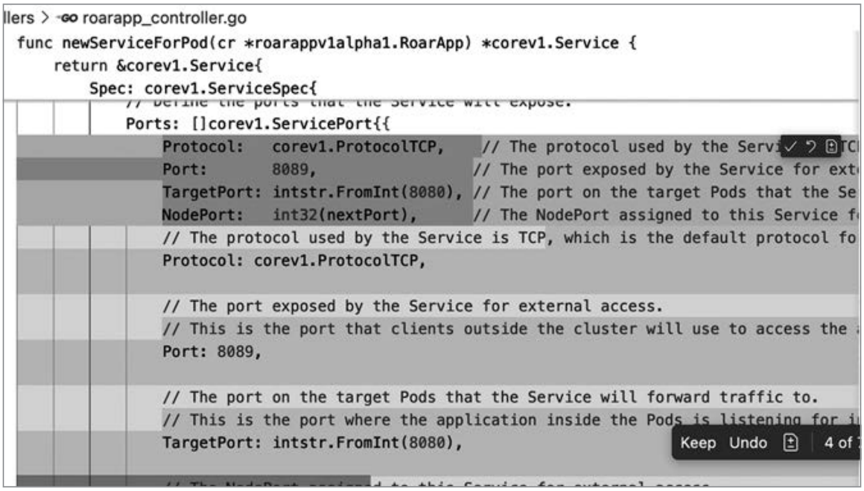


```
func newServiceForPod(cr *roarappv1alpha1.RoarApp) *corev1.Service {
    // Define a set of labels to associate the Service with the corres
    // These labels will be used by the Service to select the appropri
    labels := map[string]string{
        "app": cr.Name,
        "app": cr.Name, // The "app" label is set to the name of the R
    }

    // Create and return a new Service object.
    // The Service is configured with metadata and specifications to e
    return &corev1.Service{
        ObjectMeta: metav1.ObjectMeta{
            // The service name includes the RoarApp name and the port
            // The name of the Service is con
            // This ensures that each Service has a unique name.
            Name:      cr.Name + "-service-" + strPort,
```

Рис. 6.5. Обновленные строки после выполнения команды подробного комментирования

Уже лучше. Если вам понадобится еще больше комментариев, можно попросить Copilot дать комментарий к каждой строке в теле функции. Примерный промпт может выглядеть так: «Подробно прокомментируй тело функции, чтобы любой разработчик мог понять, что делает каждая строка».

Используя такой промпт, вы получите еще более подробные комментарии для тела функции, содержащие детальное описание некоторых элементов. Полный листинг доступен в Интернете (https://oreil.ly/rZBRV), фрагмент различий представлен на рис. 6.6.



```
func newServiceForPod(cr *roarappv1alpha1.RoarApp) *corev1.Service {
    return &corev1.Service{
        Spec: corev1.ServiceSpec{
            // Define the ports that the Service will expose.
            Ports: []corev1.ServicePort{{
                Protocol:   corev1.ProtocolTCP,    // The protocol used by the Servi
                Port:       8089,                 // The port exposed by the Service for ext
                TargetPort: intstr.FromInt(8080), // The port on the target Pods that the Se
                NodePort:   int32(nextPort),      // The NodePort assigned to this Service f
                // The protocol used by the Service is TCP, which is the default protocol fo
                Protocol: corev1.ProtocolTCP,

                // The port exposed by the Service for external access.
                // This is the port that clients outside the cluster will use to access the
                Port: 8089,

                // The port on the target Pods that the Service will forward traffic to.
                // This is the port where the application inside the Pods is listening for i
                TargetPort: intstr.FromInt(8080),

                // The NodePort assigned to this Service for external access
```

Рис. 6.6. Более подробные комментарии

Комментарии и документация

Вы могли заметить, что мы просим Copilot прокомментировать наш код, а не документировать его. Документирование в коде реализуется путем создания комментариев, поэтому в данной ситуации в промпте можно использовать любой из этих терминов. Строго говоря, когда вы просите Copilot документировать код, по умолчанию он может создать блок комментариев только в начале метода/функции, как было показано ранее.

В конечном счете можно объединить два предыдущих промпта, чтобы реализовать все изменения. Вот пример, который неплохо работает:

```
Подробно документируй заголовок и тело функции, чтобы любой разработчик мог
понять общее назначение функции и каждой строки
```

Еще одна приятная возможность документирования в Copilot — он может определять, когда в коде используется фреймворк документирования, с которым можно работать.

Генерирование документации, совместимой с фреймворком

В разных языках программирования используются разные стили документирования. Например, в коде C# есть XMLDoc, в Python — pydoc, а в JavaScript — JSDoc. Если Copilot понимает, что код написан на языке, в котором есть стандартный инструмент документирования, то по умолчанию создаст комментарии, совместимые с этим инструментом. Такая возможность может быть очень полезной для интеграции со средой разработки и другими инструментами и часто предпочтительнее простых комментариев.

Для примера возьмем определение класса Java с одним методом, который начинается так:

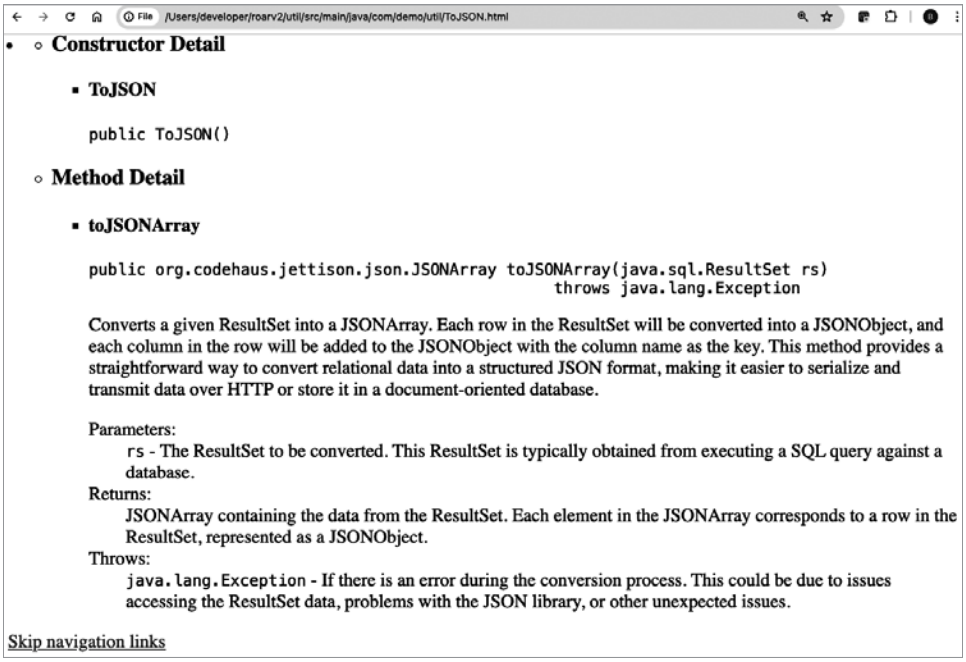
```
package com.demo.util;

import org.codehaus.jettison.json.JSONArray;
import org.codehaus.jettison.json.JSONObject;
import java.sql.ResultSet;
import org.owasp.esapi.ESAPI;

public class ToJSON {


    public JSONArray toJSONArray(ResultSet rs) throws Exception {

            JSONArray json = new JSONArray(); //Возвращаемый массив JSON
            String temp = null;

            try {
                 // Нам понадобятся имена столбцов; этот вызов сохраняет
                 // метаданные таблицы, в том числе имена столбцов.
                 java.sql.ResultSetMetaData rsmd = rs.getMetaData();
```

Если попросить Copilot документировать этот класс, то он добавит расширенный набор комментариев. Полный листинг доступен в Интернете (https://oreil.ly/LCxOH), фрагмент различий приведен ниже:

```
package com.demo.util;

// Импортирование необходимых библиотек для обработки JSON и результатов SQL
...
import java.sql.ResultSet;
// Импортирование ESAPI для целей безопасности, хотя в приведенном
// фрагменте эта возможность не используется
import org.owasp.esapi.ESAPI;

/**
* Класс ToJSON предоставляет метод для преобразования ResultSet
* из запроса к базе данных в JSONArray. Такая возможность может пригодиться
* для создания JSON API, взаимодействующих с реляционными базами данных.
...
*/
public class ToJSON {

    /**
     * Преобразует заданный объект ResultSet в JSONArray.
     * Каждая строка ResultSet преобразуется в JSONObject,
     * а каждый столбец в строке добавляется в JSONObject
...
     *
     * @param rs Преобразуемый объект ResultSet. Объект ResultSet обычно
     * возвращается в результате выполнения запроса SQL к базе данных.
```

```
     * @return Объект JSONArray с данными из ResultSet.
     * Каждый элемент в JSONArray соответствует строке ResultSet,
     * представленной в виде JSONObject.
     * @throws Exception Если в процессе преобразования произойдет ошибка.
     * Например, если...
     */
    public JSONArray toJSONArray(ResultSet rs) throws Exception {
        JSONArray json = new JSONArray(); // Инициализирует массив JSON,
                                          // который будет заполнен и возвращен
        String temp = null; // Временная строка для хранения данных,
                            // в этом фрагменте не используется
...
```

Комментарии в заголовке относятся к определению класса. Они изначально генерируются в форме, совместимой с инструментарием Javadoc. `/**`, части `@param`, `@return` и `@throw` — все эти конструкции Javadoc использует для генерирования документации HTML для класса. На рис. 6.7 показан пример документации Javadoc, сгенерированной по комментариям, добавленным Copilot.



- ◦ **Constructor Detail**

  - ▪ **ToJSON**

    `public ToJSON()`

- ◦ **Method Detail**

  - ▪ **toJSONArray**

```
public org.codehaus.jettison.json.JSONArray toJSONArray(java.sql.ResultSet rs)
                                                throws java.lang.Exception
```

Converts a given ResultSet into a JSONArray. Each row in the ResultSet will be converted into a JSONObject, and each column in the row will be added to the JSONObject with the column name as the key. This method provides a straightforward way to convert relational data into a structured JSON format, making it easier to serialize and transmit data over HTTP or store it in a document-oriented database.

Parameters:
`rs` - The ResultSet to be converted. This ResultSet is typically obtained from executing a SQL query against a database.
Returns:
JSONArray containing the data from the ResultSet. Each element in the JSONArray corresponds to a row in the ResultSet, represented as a JSONObject.
Throws:
`java.lang.Exception` - If there is an error during the conversion process. This could be due to issues accessing the ResultSet data, problems with the JSON library, or other unexpected issues.

Skip navigation links

Рис. 6.7. Документация Javadoc, сгенерированная по комментариям Copilot

Помимо создания документации для базовых фреймворков документирования, Copilot можно использовать для решения другой стандартной задачи — документирования API.

Создание документации для API

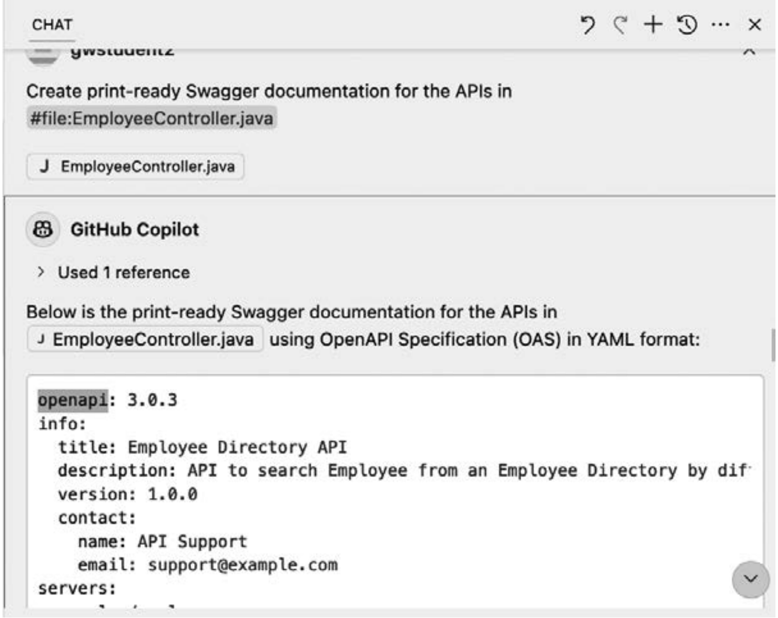
Если ваш проект использует фреймворк документирования API, такой как Swagger (https://swagger.io), то можно попросить Copilot сгенерировать соответствующую документацию Swagger для ваших API. Рассмотрим пример.

Допустим, у вас есть простое приложение для управления каталогом с информацией о работниках компании. Приложение написано на Java и уже настроено для Swagger. Можно попросить Copilot сгенерировать готовую к печати документацию Swagger для API. Для этого нужно сослаться на конкретный файл, используя такой промпт: «Создай готовую к печати документацию Swagger для API в #file:EmployeeController.java».

Готовый к печати

Мы использовали в промпте термин «готовый к печати» (print-ready), чтобы Copilot сгенерировал документацию в реальном формате (а не пытался заново создавать комментарии в коде). Можно было использовать и термин «готовый к публикации» (publish-ready).

Ответ Copilot показан на рис. 6.8, а полный вывод доступен в репозитории книги на GitHub (https://oreil.ly/Kj8Lj).



Create print-ready Swagger documentation for the APIs in #file:EmployeeController.java

Below is the print-ready Swagger documentation for the APIs in EmployeeController.java using OpenAPI Specification (OAS) in YAML format:

```
openapi: 3.0.3
info:
  title: Employee Directory API
  description: API to search Employee from an Employee Directory by dif
  version: 1.0.0
  contact:
    name: API Support
    email: support@example.com
servers:
```

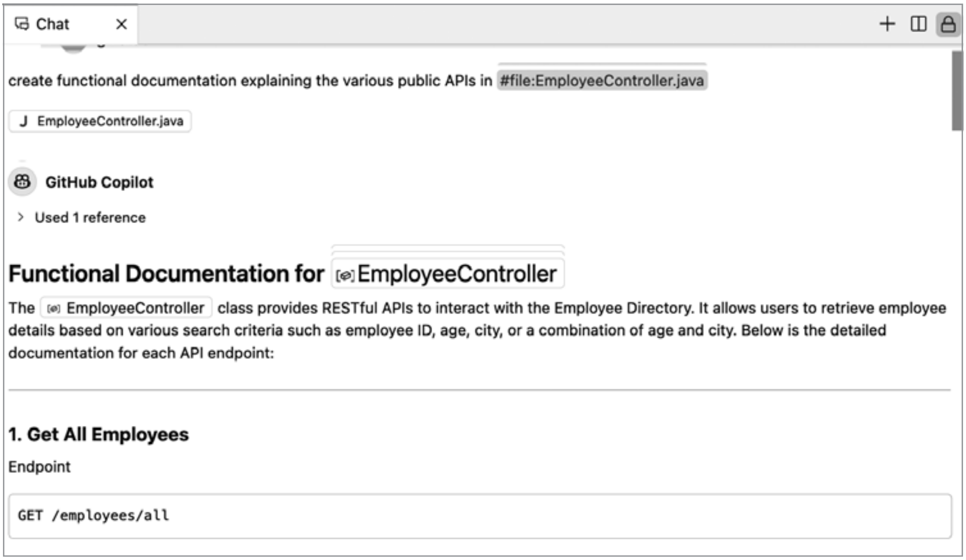
Рис. 6.8. Документация Swagger, сгенерированная на основании кода

Чтобы получить желаемый результат, необходимо (помимо дескриптора «готовый к печати») точно указать файлы для Copilot с помощью селектора `#file` (или нескольких селекторов, если вы хотите сослаться на несколько файлов). Если вы попытаетесь просто передать Copilot все рабочее пространство, используя промпт наподобие «@workspace, создай документацию Swagger для API», то обычно Copilot начнет объяснять, *как создать* документацию Swagger для вашего проекта (вместо того чтобы просто сгенерировать ее).

Создание функциональной документации

Copilot может создать функциональную документацию, предназначенную для внешних пользователей. Например, если мы используем код API из предыдущего раздела, то промпт будет примерно таким: «Создай функциональную документацию с объяснением общедоступных API в #file:EmployeeController.java».

На рис. 6.9 показано начало функциональной документации API, сгенерированной по этому промпту. Обратите внимание: в выводе приводится простое описание в удобочитаемом формате, но указываются необходимые подробности о структуре API. На рисунке это не показано, но Copilot описал набор действий, которые будет выполнять, в том числе фазы «Выявление общедоступных API», «Сигнатуры методов» и «Отображение конечных точек». Полный вывод доступен в репозитории книги на GitHub (https://oreil.ly/lV7jV).



create functional documentation explaining the various public APIs in #file:EmployeeController.java

Functional Documentation for EmployeeController

The EmployeeController class provides RESTful APIs to interact with the Employee Directory. It allows users to retrieve employee details based on various search criteria such as employee ID, age, city, or a combination of age and city. Below is the detailed documentation for each API endpoint:

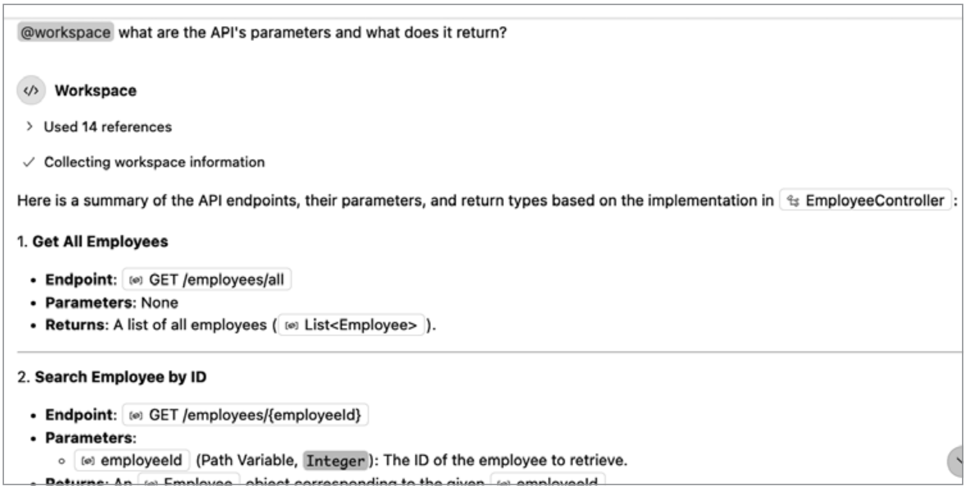
1. Get All Employees

Endpoint

```
GET /employees/all
```

Рис. 6.9. Сгенерированная функциональная документация

В целом при создании подобной документации действует такое же правило, как при генерировании документации API, которую мы обсуждали в предыдущем подразделе: в промпте нужно указать конкретные файлы вместо того, чтобы просить Copilot работать со всем рабочим пространством. Впрочем, можно задать общие вопросы, используя `@workspace`, и получить хорошую сводную документацию. Например, если вы введете промпт «@workspace, какие параметры используют API и что они возвращают?», то получите полезную сводку. Полный список доступен в Интернете (https://oreil.ly/sXKOv), а фрагмент показан на рис. 6.10.



@workspace what are the API's parameters and what does it return?

Here is a summary of the API endpoints, their parameters, and return types based on the implementation in `EmployeeController`:

1. **Get All Employees**
   - **Endpoint:** `GET /employees/all`
   - **Parameters:** None
   - **Returns:** A list of all employees (`List<Employee>`).

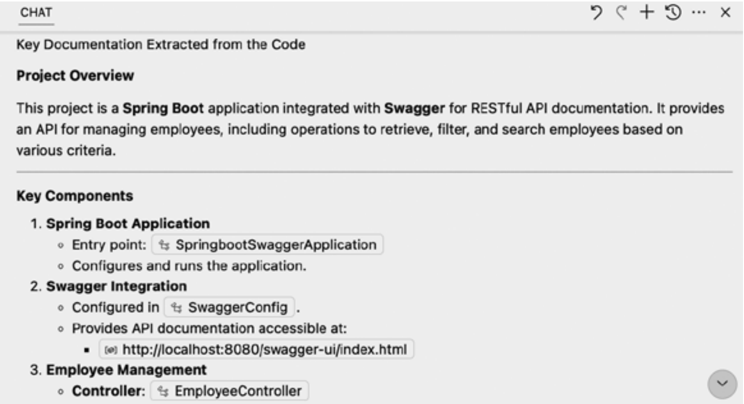
2. **Search Employee by ID**
   - **Endpoint:** `GET /employees/{employeeId}`
   - **Parameters:**
     - `employeeId` (Path Variable, `Integer`): The ID of the employee to retrieve.

Рис. 6.10. Запрос параметров API

Извлечение сводной документации из кода

В некоторых типах проектов Copilot можно использовать для извлечения сводки по ключевой документации. Такая сводка может оказаться чрезвычайно полезной, поскольку позволяет быстро получить важные подробности о кодовой базе проекта. В качестве примера используем проект Java Swagger из предыдущего подраздела. Собрать ключевую информацию о нем можно с помощью промпта «@workspace, извлеки ключевую документацию из кода».

В нашем случае Copilot извлекает данные из файлов `pom.xml` и `README.md`, а также соответствующих файлов `*.java`. Затем приводит описание ключевых элементов структуры проекта, зависимостей, конфигурации Swagger и тестирования. На рис. 6.11 приведен фрагмент, а полный список доступен на GitHub (https://oreil.ly/wSmbs).



Key Documentation Extracted from the Code

Project Overview

This project is a **Spring Boot** application integrated with **Swagger** for RESTful API documentation. It provides an API for managing employees, including operations to retrieve, filter, and search employees based on various criteria.

Key Components

1. **Spring Boot Application**
    - Entry point: `SpringbootSwaggerApplication`
    - Configures and runs the application.
2. **Swagger Integration**
    - Configured in `SwaggerConfig`.
    - Provides API documentation accessible at:
        - `http://localhost:8080/swagger-ui/index.html`
3. **Employee Management**
    - **Controller:** `EmployeeController`

Рис. 6.11. Ключевая документация, извлеченная из кода

КОПИРОВАНИЕ И ЭКСПОРТИРОВАНИЕ ДИАЛОГОВ В ЧАТЕ

Сохранить содержимое диалога в чате можно несколькими способами. Самый простой — щелкнуть правой кнопкой мыши на диалоге и выбрать команду Copy (сохраняется текущий диалог) или Copy All (сохраняются все диалоги в текущем чате).

Кроме того, можно щелкнуть на элементе управления Views and More Actions (кнопка с многоточием) в левом верхнем углу текущего чата и открыть чат в новом сеансе редактора в текущей IDE или же открыть текущий чат в отдельном окне.

Важно помнить, что при копировании данных из чата вы получаете текст в формате Markdown. Чтобы увидеть скопированный текст в таком же виде, в каком он выводится на панели чата, его необходимо сохранить в виде файла Markdown (расширение `.md`) и просмотреть в программе, которая умеет выводить файлы такого типа[1].

Если вы используете VS Code или Codespaces, то можете экспортировать диалог в формате JSON через палитру команд. Для этого откройте ее и выполните поиск по строке `Chat: Export Chat`. Сначала убедитесь, что курсор установлен в диалоге на панели чата.

На этом обзор возможностей Copilot по созданию документации завершен. Перейдем к похожей, но, возможно, более важной его функциональности — объяснению контента.

---

[1] Вы также можете воспользоваться любым онлайн-инструментом для отображения Markdown, вставив туда скопированный текст. В этом случае можно избежать создания вспомогательных временных файлов.

Объяснение контента

Помимо предоставления документации по работе кода, Copilot может давать пояснения по контенту, с которым он работает и который генерирует. Эта функциональность может быть полезной, если вы хотите:

- понять код, написанный на новом для вас языке программирования;
- понять сгенерированный код и используемые в нем соглашения;
- разобраться в логике кода;
- получить объяснение неполадок в коде;
- выяснить, что было сделано в терминале;
- понять, как выполнять операции в терминале.

В этом разделе мы кратко рассмотрим каждый из этих случаев. Для них всех характерны несколько общих моментов.

- Главную роль играет промпт. Если вы составите правильный промпт для конкретной ситуации и с его помощью объясните, что вам нужно, то Copilot сможет дать качественное объяснение.
- В промптах «объясни» — обычно один из ключевых терминов, однако он не является строго обязательным. С таким же успехом можно спросить «почему», «как» или задать другие вопросы.

Объяснение кода на незнакомом языке

Допустим, вы программируете на Python, но неожиданно вам поручили реализовать новый проект на Go. Copilot может помочь вам создать код, а также понять ключевые аспекты работы с этим языком. Например, можно ввести в Copilot такой промпт: «Я программирую на Python. Что мне нужно понимать в первую очередь о том, как писать код на Go?»

Когда я задал этот вопрос во время написания данной главы, Copilot выдал информацию о таких 12 областях, как:

- статическая типизация;
- компиляция;
- различия в синтаксисе;
- пакеты и импортирование;
- функции;
- обработка ошибок;
- конкурентность;
- структуры и методы;
- указатели;
- стандартная библиотека;
- инструменты;
- интерфейсы.

В отдельных случаях Copilot приводил короткие фрагменты кода, чтобы проиллюстрировать основные положения, и завершил объяснение фразой «Понимание этих ключевых различий поможет вам более плавно перейти с Python на Go». Полный ответ Copilot доступен на GitHub (https://oreil.ly/roJZB).

Кроме того, вы можете выбрать конкретные части кода и попросить Copilot объяснить, как они работают, — для этого используйте команду `/explain`. Вы даже можете попросить Copilot перевести существующий код на другой язык программирования. Как обычно, нужно убедиться в том, что предложения или ответы Copilot точны и актуальны для вашей ситуации.

COPILOT КАК УЧИТЕЛЬ

Один из менее распространенных способов использования Copilot для изучения новых языков — выбрать его в качестве *учителя* и попросить дать вам задачу для решения на новом языке. Рассмотрим пример.

Промпт: «Объясни, как написать функцию на Go, и дай мне задачу для выполнения». Copilot выдает ответ, в котором содержатся:

- формулировка задачи;
- последовательность действий для решения задачи;
- реальный код Go для успешного решения задачи.

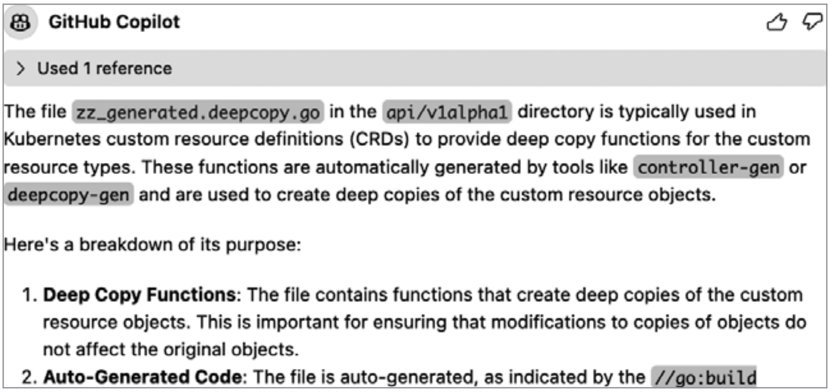
Полный ответ Copilot доступен на GitHub (https://oreil.ly/4RVS5).

Copilot не только помогает понять основы нового языка, но и выделяет используемые в нем соглашения, а также объясняет назначение кода, автоматически сгенерированного на этом языке.

Разъяснение сгенерированного кода и соглашений

Представьте, что вы новичок в Kubernetes и Go и вам нужно создать оператор Kubernetes для управления пользовательскими ресурсами. Вероятно, вы используете Operator SDK (https://oreil.ly/Lfz8O), чтобы создать первичную структуру вашего проекта на Go.

SDK автоматически генерирует служебные файлы, среди которых встречаются файлы с загадочными именами наподобие `zz_generated.deepcopy.go`. Один из первых вопросов, которые у вас могут возникнуть, — для чего нужен тот сгенерированный файл? Чтобы узнать это, просто спросите Copilot: «Для чего нужен код в #file:api/v1alpha1/zz_generated.deepcopy?» Ответ (частично показанный на рис. 6.12) начинается с общего объяснения, а затем Copilot описывает назначение кода по пунктам. Полный ответ доступен на GitHub (https://oreil.ly/I0-3j).



The file `zz_generated.deepcopy.go` in the `api/v1alpha1` directory is typically used in Kubernetes custom resource definitions (CRDs) to provide deep copy functions for the custom resource types. These functions are automatically generated by tools like `controller-gen` or `deepcopy-gen` and are used to create deep copies of the custom resource objects.

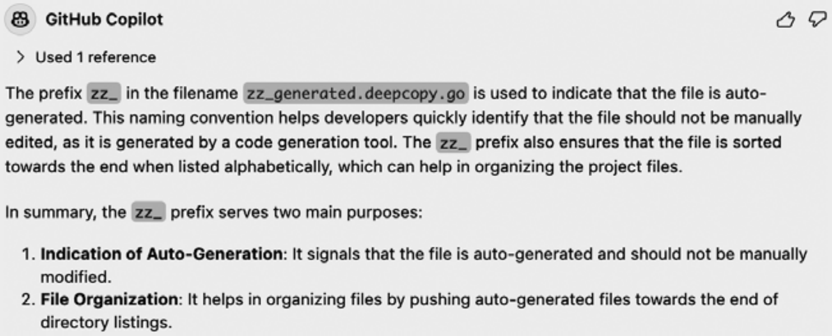
Here's a breakdown of its purpose:

1. **Deep Copy Functions**: The file contains functions that create deep copies of the custom resource objects. This is important for ensuring that modifications to copies of objects do not affect the original objects.
2. **Auto-Generated Code**: The file is auto-generated, as indicated by the `//go:build`

Рис. 6.12. Copilot объясняет назначение кода

В таких ответах Copilot обычно дает в конце вывода сводку, по которой можно быстро понять суть ответа. В данном случае Copilot отмечает: «Таким образом, zz_generated.deepcopy.go — автоматически сгенерированный файл, который предоставляет необходимые функции глубокого копирования для пользовательских типов ресурсов в Kubernetes, обеспечивая их правильную обработку и управление ими».

У вас может возникнуть вопрос, почему файлу присвоено именно такое имя. Чтобы получить от Copilot информацию о соглашениях об именах, введите такой промпт: «Почему имя файла начинается с zz_?» Copilot дает подробный ответ со списком причин для выбора имен (рис. 6.13).



The prefix `zz_` in the filename `zz_generated.deepcopy.go` is used to indicate that the file is auto-generated. This naming convention helps developers quickly identify that the file should not be manually edited, as it is generated by a code generation tool. The `zz_` prefix also ensures that the file is sorted towards the end when listed alphabetically, which can help in organizing the project files.

In summary, the `zz_` prefix serves two main purposes:

1. **Indication of Auto-Generation**: It signals that the file is auto-generated and should not be manually modified.
2. **File Organization**: It helps in organizing files by pushing auto-generated files towards the end of directory listings.

Рис. 6.13. Copilot объясняет соглашение об именах zz_

Помимо разъяснения общих соглашений и основных принципов языка, Copilot может объяснить общую логику набора кода.

Объяснение логики, используемой в коде

Обобщенное описание логики кода может быть полезно, если вы хотите:

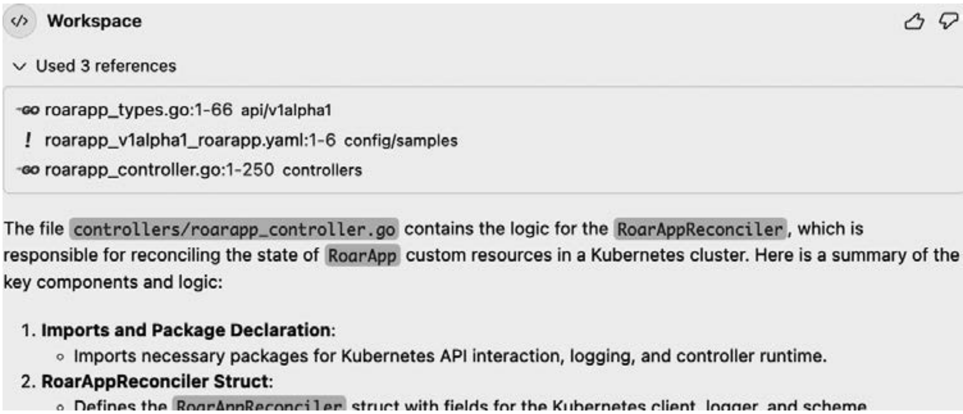
- быстро получить общее представление о чужом коде, который вам передали или который вы поддерживаете;
- изучить код, написанный на незнакомом для вас языке или с использованием незнакомого фреймворка;
- проверить уровень вашего понимания кода после его ревью;
- получить альтернативное объяснение работы кода, необходимое для выполнения ревью, pull-запросов и т. д.

Цель — получить от Copilot описание общей логики и взаимосвязанных частей (например, основных используемых компонентов) без привязки к конкретному языку. Объяснения Copilot должны иметь разговорный стиль, как если бы мы общались с другим разработчиком.

Промпты, используемые в таких случаях, могут быть разными, но я приведу пример для такого файла контроллера: «@workspace, приведи общее описание логики в #file:controllers/roarapp_controller.go».

Ответ Copilot начинается с общего объяснения того, что должен делать этот код: «Согласовать состояние... пользовательских ресурсов в кластере Kubernetes».

Затем Copilot предоставляет сводку ключевых компонентов в файле и того, что делает каждый из них: структуры данных, аннотации, функции и т. д. Фрагмент ответа показан на рис. 6.14, а полностью он доступен на GitHub (https://oreil.ly/5oG3v).



roarapp_types.go:1-66 api/v1alpha1
roarapp_v1alpha1_roarapp.yaml:1-6 config/samples
roarapp_controller.go:1-250 controllers

The file `controllers/roarapp_controller.go` contains the logic for the `RoarAppReconciler`, which is responsible for reconciling the state of `RoarApp` custom resources in a Kubernetes cluster. Here is a summary of the key components and logic:

1. **Imports and Package Declaration**:
   - Imports necessary packages for Kubernetes API interaction, logging, and controller runtime.
2. **RoarAppReconciler Struct**:
   - Defines the `RoarAppReconciler` struct with fields for the Kubernetes client, logger, and scheme

Рис. 6.14. Copilot выдает сводку о логике, используемой в файле

Как вариант, в промпте слово «логику» можно заменить «процессом» или «последовательностью операций»: например, «@workspace, приведи общее описание последовательности операций в #file:controllers/roarapp_controller.go». В этом случае ответ получается чуть более простым, но содержит похожую информацию о различных компонентах.

Изучить новый язык и понять написанный код поможет еще один интересный прием. Вместо того чтобы просить Copilot объяснить, как или что делает код, можно задать вопрос о наличии проблем в этом коде.

Разъяснение возможных неполадок в коде

Другой подход при использовании Copilot для объяснения кода — попросить выявить в нем потенциальные проблемы. Этот подход можно применить к коду, который вы написали, изучаете или «получили в наследство» от кого-то.

Возвращаясь к примеру с пользовательскими ресурсами Kubernetes, приведенному выше, можно спросить у Copilot, какие неполадки могут возникнуть в коде, реализующем контроллер в редакторе. Для этого можно использовать такой промпт: «@workspace, объясни возможные проблемы с кодом в #file:controllers/roarapp_controller.go».

Использование @workspace и #file

Напомню, что переменные #selection или #file в промпте помогают Copilot сосредоточиться на контексте, который он должен учитывать при формировании ответа. Но в некоторых случаях приходится также использовать идентификатор @workspace, чтобы Copilot принял во внимание полную картину того, что может повлиять на выполнение кода.

Copilot дает подробный ответ с перечислением областей, на которые стоит обратить особое внимание, в том числе:

- обработка ошибок;
- конфликты ресурсов;
- логика возвращения в очередь;
- освобождение ресурсов;
- проблемы конкурентности;
- ведение журнала;
- проверка данных;
- разрешения RBAC.

Полный ответ доступен в репозитории GitHub (https://oreil.ly/v7I75).

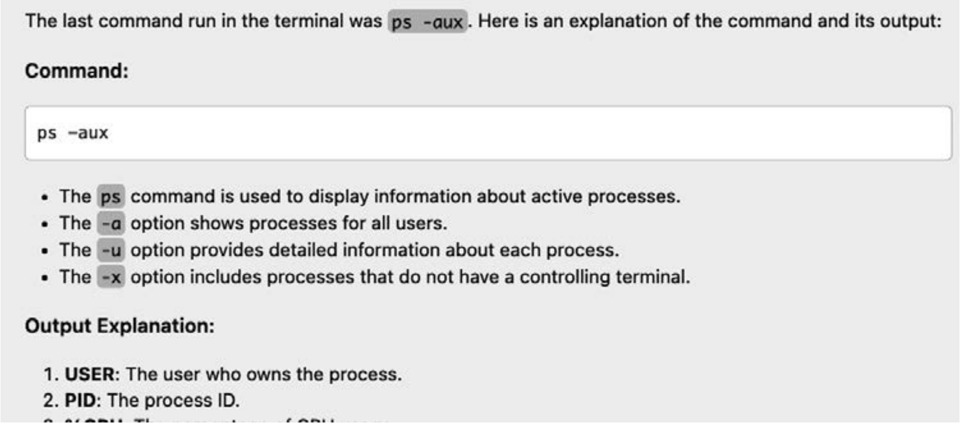
Мы рассмотрели разные способы, с помощью которых Copilot может объяснить, что происходит в коде. Напоследок обсудим, как Copilot может помочь с объяснением происходящего в другом контексте — в терминале.

Объяснение кода в терминале

Copilot может обратиться к терминалу, чтобы получить информацию о последней команде, с помощью ключевого слова `#terminalLastCommand`. Допустим, последней была выполнена команда `ps`:

```
@techupskills → /workspaces/op-new (main) $ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
codespa+       1  0.0  0.0   1136   640 ?        Ss   14:30   0:00
 /sbin/docker-init
codespa+       8  0.0  0.0   7236  1920 ?        S    14:30   0:00
 sleep infinity
root          23  0.0  0.0  12196  3484 ?        Ss   14:30   0:00
 sshd: /usr/sbin/s
codespa+     338  0.0  0.0   2616  1536 ?        Ss   14:30   0:00
 /bin/sh
```

Если вы хотите получить более подробную информацию о том, что делает эта команда, то можете ввести в Copilot промпт «Объясни #terminalLastCommand». Как показано на рис. 6.15, Copilot определяет, какая команда была последней, и предоставляет объяснение ее самой, ее входных параметров и результатов.



The last command run in the terminal was `ps -aux`. Here is an explanation of the command and its output:

Command:

```
ps -aux
```

- The `ps` command is used to display information about active processes.
- The `-a` option shows processes for all users.
- The `-u` option provides detailed information about each process.
- The `-x` option includes processes that do not have a controlling terminal.

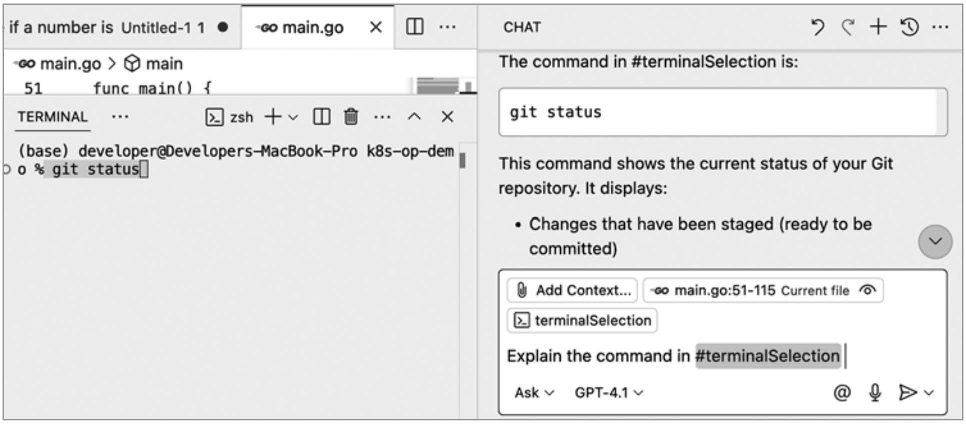
Output Explanation:

1. **USER**: The user who owns the process.
2. **PID**: The process ID.

Рис. 6.15. Copilot объясняет последнюю команду в терминале

Полный ответ доступен на GitHub (https://oreil.ly/yCMOO).

Кроме того, Copilot может объяснить выбранное содержимое терминала. Например, если ранее была выполнена команда `git status`, то вы можете выделить ее и ввести такой промпт: «Объясни команды в #terminalSelection». Copilot выдает информацию по этой теме (рис. 6.16).



```
(base) developer@Developers-MacBook-Pro k8s-op-dem
o % git status
```

The command in #terminalSelection is:

```
git status
```

This command shows the current status of your Git repository. It displays:

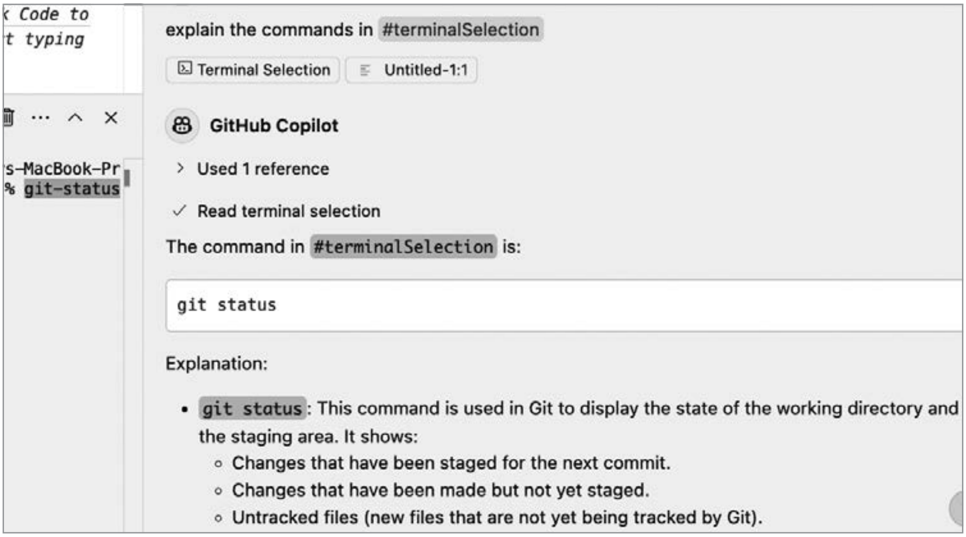
- Changes that have been staged (ready to be committed)

Explain the command in #terminalSelection

Рис. 6.16. Объяснение команды, выделенной в терминале

Copilot может объяснить не только команду. Вы можете выделить любой текст, выводимый в терминале, — например, результат работы команды. Допустим, вы выделили текст `Your branch is up to date with 'origin/main'` в выводе команды `git status`. Идентификатор уже ссылается на любое выделенное содержимое в терминале, поэтому промпт можно сократить до «Объясни #terminalSelection».

Copilot объясняет смысл выделенной фразы. На рис. 6.17 изображен фрагмент ответа для этого промпта.



explain the commands in #terminalSelection

The command in #terminalSelection is:

```
git status
```

Explanation:

- `git status`: This command is used in Git to display the state of the working directory and the staging area. It shows:
  - Changes that have been staged for the next commit.
  - Changes that have been made but not yet staged.
  - Untracked files (new files that are not yet being tracked by Git).

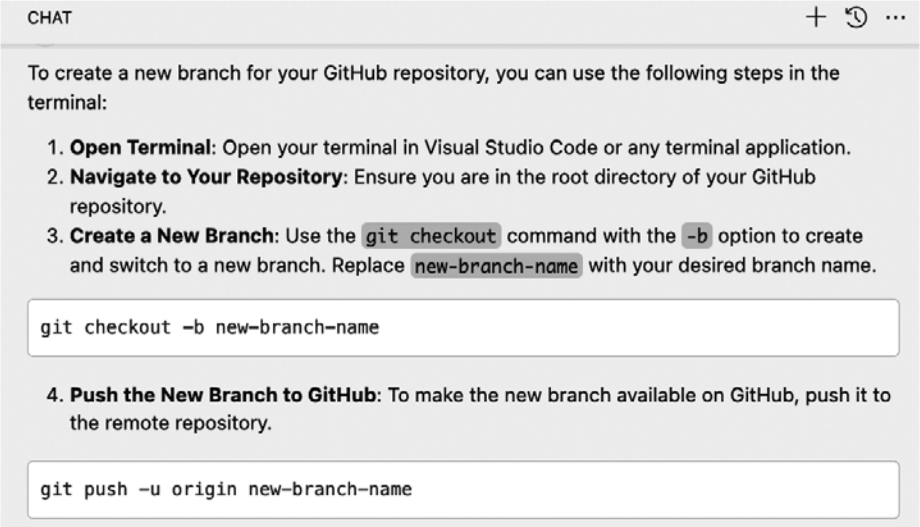
Рис. 6.17. Объяснение команды git status, выделенной в терминале

Кроме того, вы можете попросить Copilot объяснить, *как* что-то сделать из терминала. Об этом и поговорим далее.

Объяснение выполнения операций в терминале

Copilot может описать процесс и указать названия команд для выполнения операций в терминале. Представьте, что хотите спросить у Copilot, как создать новую ветку в репозитории GitHub. Вы можете использовать контекст `@terminal` и ввести такой промпт: «@terminal, как создать новую ветку в моем репозитории GitHub?»

Copilot в разговорном стиле опишет процесс и укажет реальные команды, которые можно выполнить в терминале. На рис. 6.18 приведен фрагмент ответа, а полную версию можно просмотреть на GitHub (https://oreil.ly/0t1il).



To create a new branch for your GitHub repository, you can use the following steps in the terminal:

1. **Open Terminal**: Open your terminal in Visual Studio Code or any terminal application.
2. **Navigate to Your Repository**: Ensure you are in the root directory of your GitHub repository.
3. **Create a New Branch**: Use the `git checkout` command with the `-b` option to create and switch to a new branch. Replace `new-branch-name` with your desired branch name.

```
git checkout -b new-branch-name
```

4. **Push the New Branch to GitHub**: To make the new branch available on GitHub, push it to the remote repository.

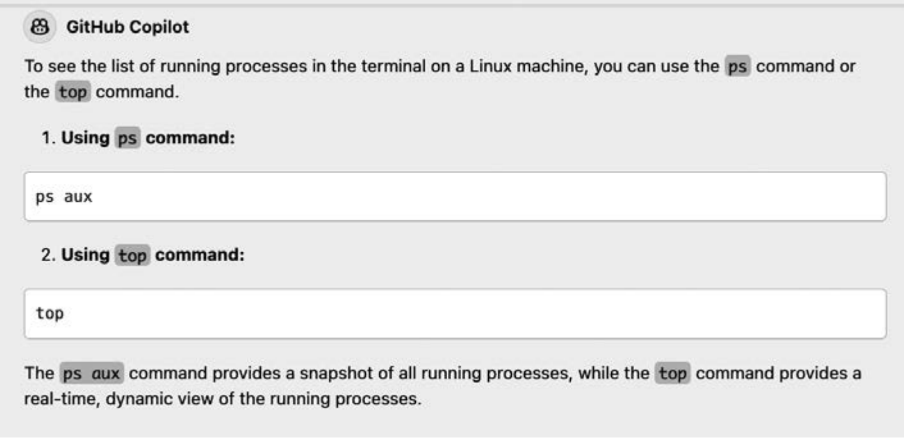
```
git push -u origin new-branch-name
```

Рис. 6.18. Объяснение процесса создания новой ветки репозитория из терминала

Аналогичный результат

Такой же результат можно получить с помощью промпта, в котором упоминается командная строка: «Как создать новую ветку в моем репозитории GitHub из командной строки?»

Этот метод применим и в ситуациях, когда вы не знаете, какую команду нужно использовать. Например, если ввести запрос «@terminal, как мне просмотреть список выполняемых процессов?», то Copilot выдаст несколько вариантов команд, которые можно выполнить из терминала (рис. 6.19).



GitHub Copilot

To see the list of running processes in the terminal on a Linux machine, you can use the `ps` command or the `top` command.

1. **Using `ps` command:**

```
ps aux
```

2. **Using `top` command:**

```
top
```

The `ps aux` command provides a snapshot of all running processes, while the `top` command provides a real-time, dynamic view of the running processes.

Рис. 6.19. Ответ Copilot о том, как просмотреть выполняемые процессы из терминала

Резюме

В этой главе мы исследовали, как Copilot может создавать документацию и давать объяснения. На базовом уровне он может документировать исходный код, вставляя в него комментарии. С помощью команды `/doc` обычно можно получить заголовочную документацию. Можно попросить Copilot сгенерировать более обширные комментарии, используя промпты с конкретными указаниями, — например, «подробно прокомментируй». Этот прием можно использовать при работе с Copilot в целом: если вы не получили желаемое, то попробуйте уточнить и расширить промпт. Если предложенные изменения вам не подойдут, то вы можете отклонить их и продолжать переформулировать промпт.

Помимо комментариев, Copilot может создавать более обширную документацию для конкретных сценариев использования. Например, он может сгенерировать документацию API и функциональную документацию, которую (после проверки) будут использовать другие разработчики. Этот подход позволяет сэкономить немало времени по сравнению с созданием такой документации менее автоматизированными способами.

Помимо документирования, Copilot может давать объяснения о работе различных компонентов кодовой базы. Благодаря этой функциональности вы можете узнать об общих концепциях и принципах, необходимых для использования нового языка или фреймворка, а также получить информацию о соглашениях, которые вам могут быть неизвестны.

Но важнее всего то, что Copilot может объяснить, как работает логика и внутренняя реализация блока кода, и выдать сводку. Это может быть особенно полезно, если вы изучаете незнакомый код или если занимаетесь его сопровождением.

Объяснения можно использовать еще одним, менее распространенным способом: попросить Copilot выполнить ревью кода, чтобы выявить потенциальные проблемы и недостатки. Этот вариант может быть очень полезным, если вы хотите оценить слабые места в коде и убедиться, что все уязвимости в нем исправлены.

Наконец, мы поговорили о том, как использовать функциональность объяснения Copilot для работы с терминалом. Вам доступны два варианта действий. В первом вы можете выделить часть вывода или дать ссылку на последнюю команду терминала, используя специальный идентификатор, и попросить Copilot объяснить ее смысл. Во втором варианте вы можете спросить о том, как выполнить в терминале ту или иную операцию, — и Copilot выдаст подробное объяснение и укажет точные названия необходимых команд.

Итак, вы увидели, насколько полезными и широкими могут быть возможности Copilot. Но важно помнить, что результаты, которые он выдает, могут быть неточными или устаревшими. Это объясняется тем, что актуальность его базы знаний зависит от свежести данных, на которых обучались используемые им модели.

В следующей главе вы узнаете, что делать с предложениями и ответами Copilot, основанными на устаревшей информации, и как действовать в подобных ситуациях.

ГЛАВА 7

Поддержание актуальности и релевантности ответов Copilot

Диапазон функциональности и форм взаимодействия с пользователем, предоставляемых GitHub Copilot, весьма впечатляет. Но ответы, которые он генерирует, порой оказываются не очень убедительными (и менее полезными), если основываются на устаревшей и нерелевантной информации. Каким бы замечательным ни был ИИ, время от времени приходится принимать дополнительные меры, чтобы направить его на нужный путь.

Например, иногда нужно привлечь внимание Copilot к некоторым частям контента, чтобы получить более точные ответы. А может, вы захотите повысить качество его обучения, чтобы ответы были более релевантными для вашей кодовой базы. И еще вам стоит обращать больше внимания на предложения Copilot — они могут быть устаревшими или ссылаться на функции, которые уже не поддерживаются.

В этой главе вы узнаете, какие стратегии и функциональность можно использовать, чтобы повысить релевантность и актуальность взаимодействия с Copilot. Мы рассмотрим приемы, которые можно применять в ситуациях, когда Copilot не знает о последних изменениях или ссылается на неподходящие источники.

Все эти меры нужны для того, чтобы помочь Copilot получить правильный контекст для предоставления наиболее точных ответов. В этой главе мы рассмотрим следующие темы:

- источники контекста;
- факторы, влияющие на актуальность и релевантность результатов;
- пользовательские стратегии улучшения работы Copilot;
- добавление контекста для повышения релевантности кода.

Само определение термина «генеративный» подразумевает создание нового контента, частично основанного на сборе и обработке контекста. Для начала вспомним, откуда берется контекст, используемый Copilot.

Источники контекста

Как я уже говорил, в IDE Copilot извлекает контекст непосредственно из рабочей среды, а именно из файлов, имеющихся в рабочем пространстве, и связанного с ними содержимого. Это могут быть такие типичные атрибуты, как имя файла, комментарии в нем, а также код до и после курсора. Помимо этого, Copilot берет в качестве контекста и более динамичные взаимодействия — например, учитывает, приняли вы или отклонили последний предложенный им код. Автоматически создавая индекс вашего репозитория, Copilot получает более полное представление о вашем проекте. Кроме того, вы можете настроить Copilot так, чтобы он учитывал другой контент.

Чтобы быстро сгенерировать предложения кода в IDE/редакторе, Copilot использует стратегию выбора текущих открытых файлов. Это позволяет ему быстро получить представление о том, что предположительно является самым важным (так как эти файлы находятся в прямой доступности).

В этом *встроенном* режиме Copilot использует несколько стратегий для обеспечения быстрого отклика. Клиент (расширение, установленное в вашем интерфейсе) запрашивает у модели небольшое количество предложений (от одного до трех). Copilot активно кэширует результаты и адаптирует предложения, если вы продолжите вводить текст. Вдобавок Copilot использует встроенные проверки, блокируя отправку запросов, если вы вводите символы, например, в середине строки — если справа от курсора нет пробелов. Затем индекс (который мы упомянули выше) обновляется на основании изменений.

Кроме того, в этом режиме формируется оценка для определения того, полезен ли промпт (то есть стоит ли он того, чтобы передавать его модели). Решение зависит от таких факторов, как предыдущие реакции пользователя на предложения. Это помогает исключить повторяющиеся инструкции или те, которые уже есть в коде.

Большинство из этих стратегий не нужны при переходе в чат-интерфейс. В этом случае у Copilot появляется больше времени на подготовку ответов, поэтому клиент может запрашивать больше вариантов завершения. Кроме того, при работе в чат-интерфейсе можно более конкретно указывать источники контекста. Copilot содержит такие конструкции, как чат-переменные и участники (о которых мы говорили в главе 3 и других главах), помогающие ему сосредоточиться на конкретных элементах.

Независимо от того, где работает пользователь, клиент собирает контекст, обрабатывает его, преобразуя в промпт, и передает выбранной вами модели. После быстрой проверки информации, возвращенной моделью, и генерации набора ответов они выводятся пользователю. Они могут содержать различные варианты завершения кода в IDE.

Упомянутая в абзаце выше *быстрая проверка* подразумевает простой поиск в ответе закономерностей, которые повторяются или могут указывать на дефект безопасности (например, жестко запрограммированные учетные данные). Эти проверки полезны, но не исключают наличия других факторов, которые могут привести к снижению релевантности и актуальности. Рассмотрим некоторые из них.

Факторы, влияющие на актуальность и релевантность

Когда вы впервые открываете чат-интерфейс Copilot, на экране появляется предупреждение:

```
Я работаю на основе ИИ, поэтому возможны ошибки. Внимательно проверьте весь
сгенерированный код и предложения и дайте обратную связь, чтобы я мог учиться
и совершенствоваться.
```

Это предупреждение напоминает о том, что результаты генеративного ИИ могут быть неточными. В конечном счете именно пользователь должен убедиться в том, что сгенерированный контент является правильным и релевантным. (И именно поэтому вы всегда должны использовать стандартные методы, такие как тестирование и ревью кода, для проверки любого контента, созданного с помощью ИИ.)

Упомянуть об этом необходимо, поскольку на актуальность и релевантность предложений и ответов Copilot могут повлиять различные факторы. К ним относятся:

- порог обучения;
- галлюцинации;
- отсутствие проверки в реальном времени;
- ошибочно выбранный контекст;
- отсутствие контекста.

Рассмотрим все эти факторы более подробно.

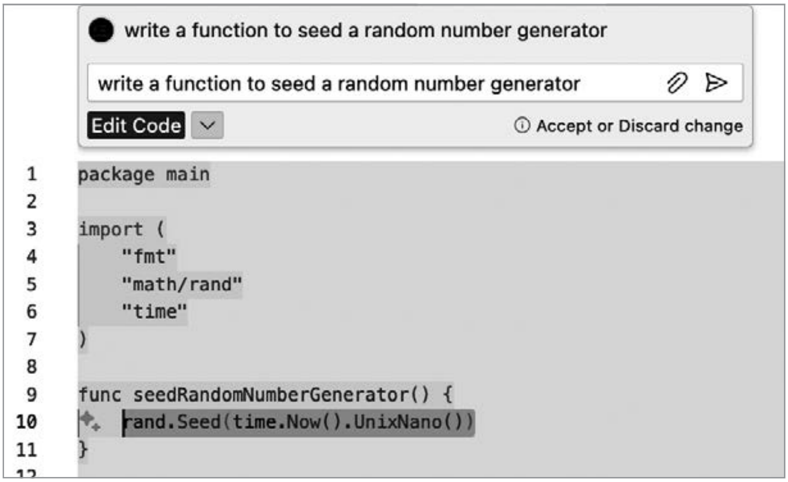
Порог обучения

LLM обучаются на данных, актуальных на момент обучения. Впоследствии модель не обладает *встроенным* знанием изменений, произошедших в областях, данные которых использовались при ее обучении. Чем больше времени прошло с момента обучения, тем выше вероятность того, что некоторые ответы и предложения Copilot окажутся неактуальными. Это может проявляться в том, что он генерирует устаревший код или код, более подходящий для предыдущей версии фреймворка или языка. И конечно, результаты будут различаться в зависимости от выбранной вами модели.

Частота обучения

Почему модели не обновляются чаще? Важно понимать, что для любого обучения им требуются миллиарды параметров и огромные наборы данных, обработка которых требует значительных затрат вычислительных ресурсов и времени. Таким образом, обновление обучающей выборки стоит дорого.

Рассмотрим простой пример проблем, которые могут возникнуть из-за порога обучения (проявлявшихся в Copilot на момент написания книги). Мы просим Copilot написать функцию для инициализации генератора случайных чисел на Go (рис. 7.1).



write a function to seed a random number generator

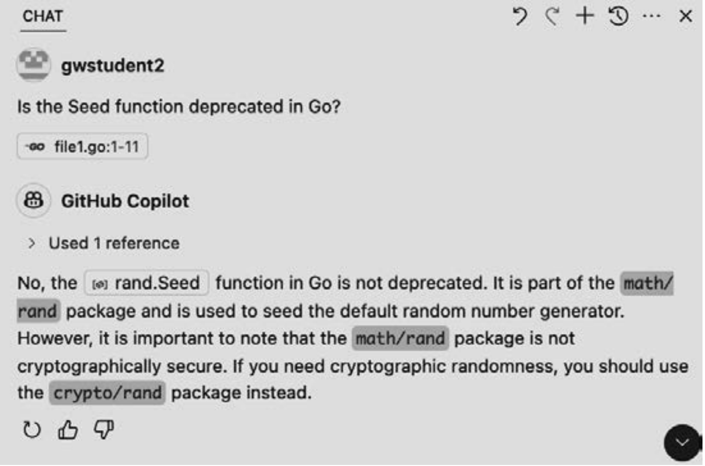
write a function to seed a random number generator

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func seedRandomNumberGenerator() {
    rand.Seed(time.Now().UnixNano())
}
```

Рис. 7.1. Copilot получает запрос на создание генератора случайных чисел на Go

Код выглядит корректным и синтаксически верным, но есть проблема. Функция `Seed` считается устаревшей в Go (https://oreil.ly/D2hpj) с версии 1.20. Посмотрим, понимает ли это Copilot. Спросите в интерфейсе Chat: «Считается ли функция Seed устаревшей в Go?» Ответ на рис. 7.2 показывает, что модель не обладает такой информацией.

Таким образом, хотя использовать функцию не рекомендуется, Copilot об этом не знает. Этот пример показывает, как работает порог обучения. Некоторые способы решения подобных проблем мы рассмотрим далее в этой главе.



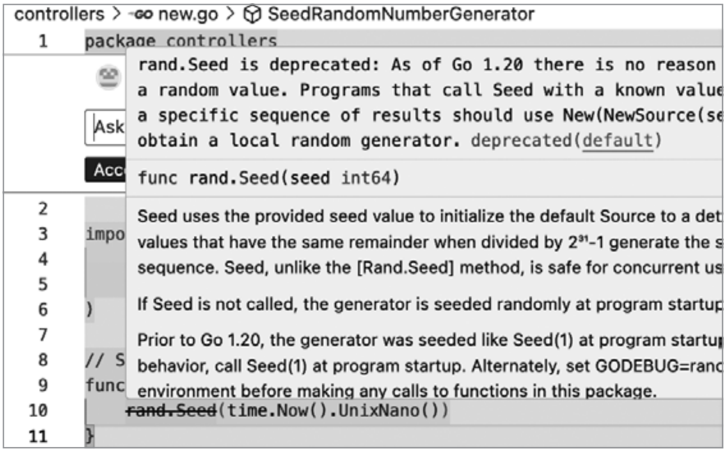
Is the Seed function deprecated in Go?

No, the `rand.Seed` function in Go is not deprecated. It is part of the `math/rand` package and is used to seed the default random number generator. However, it is important to note that the `math/rand` package is not cryptographically secure. If you need cryptographic randomness, you should use the `crypto/rand` package instead.

Рис. 7.2. Copilot отвечает на вопрос об устаревании функции Seed

ОБНАРУЖЕНИЕ УСТАРЕВШЕГО СОДЕРЖИМОГО

В некоторых IDE другие установленные программы могут обнаруживать устаревший код и сообщать о нем. На рис. 7.3 показан пример обнаружения устаревшего кода другим инструментом и соответствующее сообщение.



```
package controllers
impo
)
// S
func
    ~~rand.Seed~~(time.Now().UnixNano())
}
```

rand.Seed is deprecated: As of Go 1.20 there is no reason
a random value. Programs that call Seed with a known value
a specific sequence of results should use New(NewSource(se
obtain a local random generator. deprecated(default)

`func rand.Seed(seed int64)`

Seed uses the provided seed value to initialize the default Source to a det
values that have the same remainder when divided by 2^{31}-1 generate the s
sequence. Seed, unlike the [Rand.Seed] method, is safe for concurrent us

If Seed is not called, the generator is seeded randomly at program startup

Prior to Go 1.20, the generator was seeded like Seed(1) at program startu
behavior, call Seed(1) at program startup. Alternately, set GODEBUG=ranc
environment before making any calls to functions in this package.

Рис. 7.3. Обнаружение устаревшего кода другим инструментом

В таких случаях вы *сможете* использовать функцию исправления Copilot для генерации альтернативного кода.

Несвоевременные результаты представляют одну категорию проблем: результаты, действительные в какой-то период, перестали быть таковыми. Проблемы из другой категории возникают, когда результаты кажутся действительными для текущего контекста, но могут не отражать реальные данные или содержат ссылки на несуществующие элементы. Такие явления называются *галлюцинациями*.

Галлюцинации

LLM иногда создают ответы с данными, которые кажутся действительными, но это впечатление ошибочно. Такие ответы обозначаются термином «галлюцинации». В обычных чат-интерфейсах в качестве галлюцинаций можно рассматривать ссылки на несуществующие элементы или утверждения, которые выдаются за истинные, хотя являются ложными. В критически важных системах, таких как системы обнаружения угроз, ИИ-модель может пометить как угрозу нечто безопасное.

В контексте программирования галлюцинации могут относиться к разным категориям, таким как:

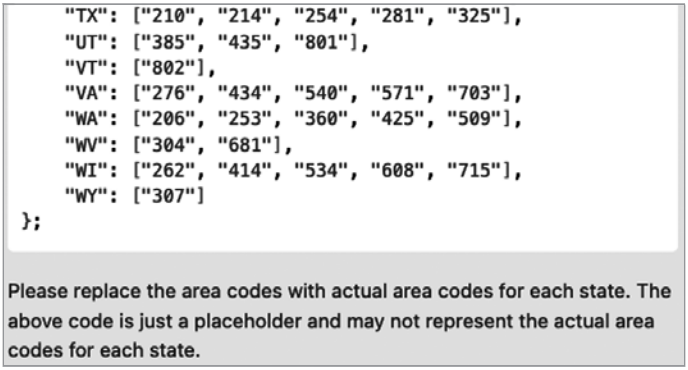
- предложения кода, в котором упоминаются идентификаторы (имена, переменные, константы и т. д.) или другой код, которого попросту нет;
- сгенерированные данные, которые могут быть неточными;
- несуществующие функции или методы;
- некорректное использование библиотек или API (или использование библиотек или API, которых вообще нет);
- вымышленные типы или классы;
- правдоподобная, но неправильная бизнес-логика;
- ошибочные предположения об окружающем коде;
- тесты, которые на самом деле не проверяют предполагаемое поведение.

Галлюцинации первой категории, безусловно, все еще могут встречаться в результатах, выдаваемых Copilot. Но сейчас они встречаются реже, чем в первые годы его существования, поскольку базовые модели улучшаются. Такие галлюцинации (ссылки на несуществующие идентификаторы или код) становятся более очевидными и обычно быстро обнаруживаются пользователями при проверке результата. Если галлюцинации будут упущены и приняты, то, скорее всего, компиляторы, синтаксические анализаторы или другие инструменты, проверяющие синтаксис, в какой-то момент обнаружат и пометят их.

Галлюцинации второй категории (связанные со сгенерированными данными) менее очевидны. Они встречаются скорее из-за отсутствия автоматической проверки сгенерированных данных в реальном времени.

Отсутствие проверки в реальном времени

Сгенерированные данные, возвращаемые Copilot, могут быть неточными. Иногда он сам напоминает вам об этом, как показано на рис. 7.4. В данном случае мы попросили его сгенерировать сопоставление телефонных кодов регионов с штатами. Сделав это, Copilot добавляет дисклеймер: «Приведенный выше код является условным и не предоставляет реальные телефонные коды регионов для всех штатов».



```
    "TX": ["210", "214", "254", "281", "325"],
    "UT": ["385", "435", "801"],
    "VT": ["802"],
    "VA": ["276", "434", "540", "571", "703"],
    "WA": ["206", "253", "360", "425", "509"],
    "WV": ["304", "681"],
    "WI": ["262", "414", "534", "608", "715"],
    "WY": ["307"]
};
```

Please replace the area codes with actual area codes for each state. The above code is just a placeholder and may not represent the actual area codes for each state.

Рис. 7.4. Оговорка Copilot о точности сгенерированных данных

Почему он добавляет этот дисклеймер? По четырем причинам.

- Данные, на которых обучается Copilot, могут содержать как точные, так и неточные фактические значения.
- Copilot генерирует результаты, соответствующие синтаксису и структуре кода.
- Copilot не запускает сгенерированный им код и не проверяет логику.
- Copilot не содержит механизмов для поиска и проверки данных на точность и актуальность.

Первые три пункта понятны без объяснений. В четвертом пункте сообщается, что у Copilot нет механизма перекрестной проверки сгенерированных результатов по другим источникам (например, в Интернете). Некоторые ИИ-методы могут использовать соответствующие инструменты, но они не интегрируются с Copilot для проверки ответов модели. Данные могут быть правильными, но гарантий нет. Короче говоря, Copilot не может определить, истинно или ложно какое-либо утверждение.

По этой причине не стоит заранее доверять информации, которую выдал Copilot. Сгенерированные им данные лучше подходят для контента, который должен иметь правильную форму, но не обязательно должен содержать правильные значения, — например, для тестовых сценариев.

Расширения Copilot

Вы можете создать расширения Copilot, которые обращаются с вызовами к API или другим инструментам, чтобы получить более актуальную обновленную информацию для конкретных ситуаций. Вы сможете найти существующее расширение, которое уже делает то, что вам нужно, либо создать собственное. Механизм Copilot Extensions подробно описывается в главе 10.

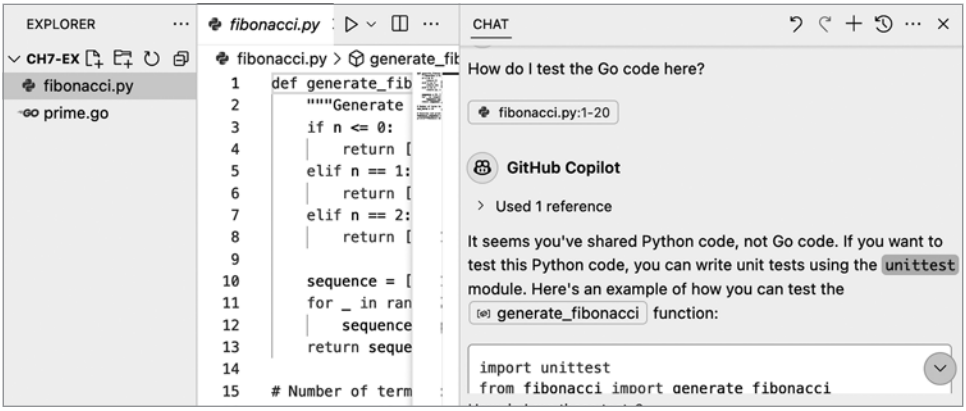
Еще одна проблема, с которой вы можете столкнуться, — когда Copilot в качестве основы для ответов берет контекст, который вообще не должен был использоваться для этого.

Ошибочно выбранный контекст

Как правило, когда вы общаетесь с Copilot в чат-интерфейсе IDE, у вас в редакторе есть активный файл. Возможно, часть его выделена. В таких случаях Copilot считает выделение главным контекстом, к которому относятся все неоднозначные вопросы (например, «Как мне протестировать это?»).

Возможно, вы хотели, чтобы Copilot использовал именно этот конкретный контент. А может, он должен сгенерировать ответ на основе другого файла или другой части активного файла. Если Copilot ошибочно выберет другой контекст, то его ответы могут вам не подойти.

Представьте, что используете несколько языков или фреймворков в проекте и заданный контекст не соответствует вашему промпту. Такой проект показан на рис. 7.5. В нем есть файл Python (`fibonacci.py`) и файл, написанный на Go (`prime.go`), как показано в левой верхней части IDE.



```
def generate_fib
    """Generate
    if n <= 0:
        return [
    elif n == 1:
        return [
    elif n == 2:
        return [

    sequence = [
    for _ in ran
        sequence
    return seque

# Number of term
```

How do I test the Go code here?

fibonacci.py:1-20

GitHub Copilot

Used 1 reference

It seems you've shared Python code, not Go code. If you want to test this Python code, you can write unit tests using the `unittest` module. Here's an example of how you can test the `generate_fibonacci` function:

```
import unittest
from fibonacci import generate_fibonacci
```

Рис. 7.5. Указание неправильного контекста для промпта

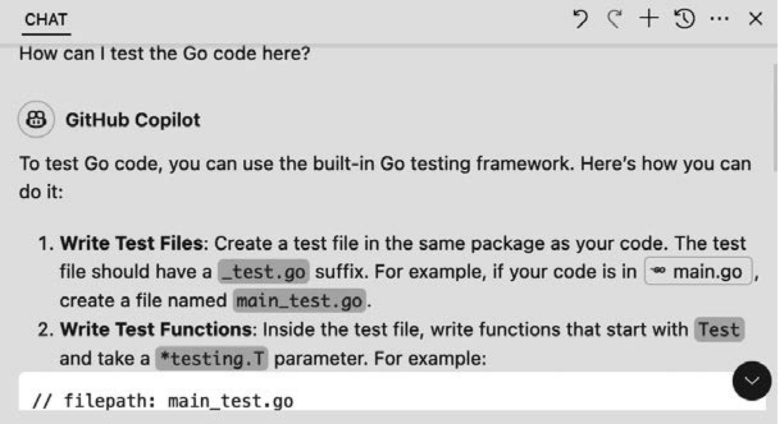
Выделив файл Python (который используется в качестве контекста), вы спрашиваете Copilot: «Как мне протестировать этот код Go?» Copilot выдает сообщение о несоответствии контекста и промпта. Затем выводит информацию о том, как выполнить тестирование Python в соответствии с активным файлом.

Еще одна разновидность проблем может возникнуть при отсутствии контекста.

Отсутствие контекста

Если вы отправляете запрос, а в рабочем пространстве нет открытых файлов и в запросе не указаны другие источники для контекста, то Copilot обычно дает более общий ответ. Например, у вас есть рабочее пространство с кодом на Go для реализации оператора Kubernetes, но при этом нет активных открытых файлов и ничего не выделено. Теперь вы пишете в чате промпт для Copilot: «Как мне протестировать этот код Go?»

Контекст не указан, поэтому Copilot либо предложит выбрать его, либо даст общий ответ. На рис. 7.6 показан пример ответа для ситуации, описанной в предыдущем абзаце. В данном случае Copilot дает общие рекомендации о том, как тестировать код Go.



How can I test the Go code here?

GitHub Copilot

To test Go code, you can use the built-in Go testing framework. Here's how you can do it:

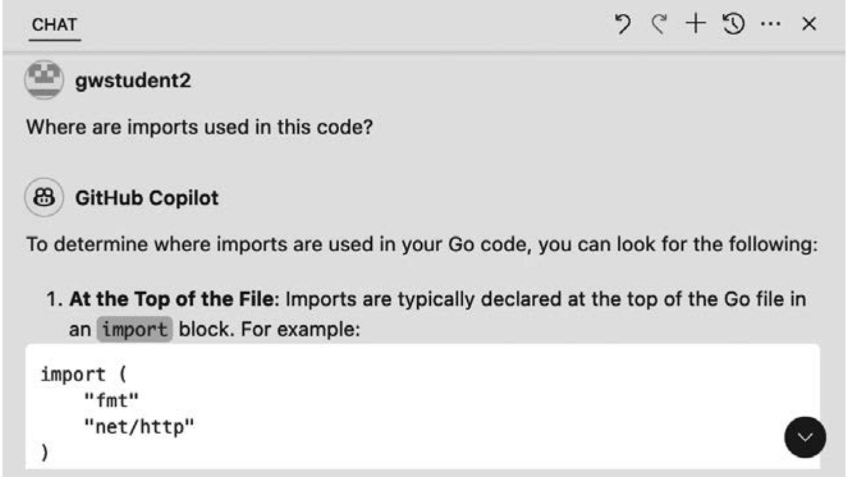
1. **Write Test Files**: Create a test file in the same package as your code. The test file should have a `_test.go` suffix. For example, if your code is in main.go, create a file named `main_test.go`.
2. **Write Test Functions**: Inside the test file, write functions that start with `Test` and take a `*testing.T` parameter. For example:

```
// filepath: main_test.go
```

Рис. 7.6. Общий совет при отсутствии контекста

Такие ответы Copilot может выдавать и для других видов запросов — например, если вы спросите, где в кодовой базе используются импортированные элементы (рис. 7.7).

Итак, при отсутствии неявного (открытые файлы) или явного контекста (ссылки в промпте) Copilot по умолчанию выдает общие ответы.



gwstudent2

Where are imports used in this code?

GitHub Copilot

To determine where imports are used in your Go code, you can look for the following:

1. **At the Top of the File:** Imports are typically declared at the top of the Go file in an `import` block. For example:

```
import (
    "fmt"
    "net/http"
)
```

Рис. 7.7. Другой пример общего ответа

Поработав с Copilot какое-то время, вы непременно столкнетесь с подобными проблемами. Степень их влияния будет зависеть от конкретной ситуации. К счастью, существует несколько стратегий по предотвращению таких сложностей.

Пользовательские стратегии улучшения работы Copilot

В целом, если ответы Copilot не слишком точны или неактуальны, для исправления этой ситуации можно использовать три стратегии:

- явное указание источника контекста;
- смену модели;
- добавление данных, которые Copilot может использовать как контекст.

Далее мы рассмотрим все три варианта.

Явное указание источника контекста

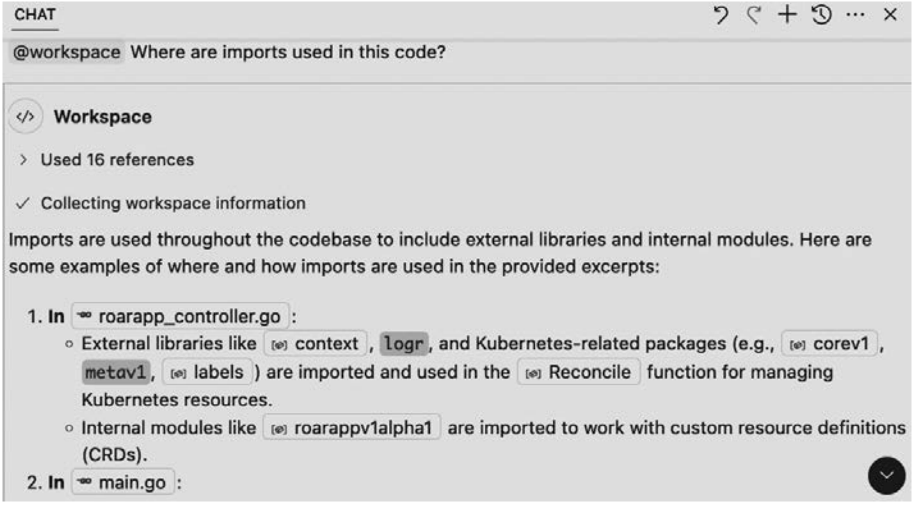
Как я упоминал ранее в этой главе, при отсутствии более явных указаний Copilot пытается собрать контекст из того, что активно используется в редакторе. Если вы работаете непосредственно в редакторе, то в большинстве случаев такое решение будет правильным. Но если вы работаете в чат-интерфейсе, то, возможно, захотите, чтобы Copilot сосредоточился на других частях проекта или отвечал на общие вопросы наподобие «Где используется ...?».

Поскольку вы работаете в чат-интерфейсе, то можете использовать такие встроенные возможности Copilot, как участники чата и чат-переменные. Я описывал их подробно в главе 3. Здесь я просто напомню, как с их помощью решать возможные проблемы.

Участники чата обладают информацией о разных областях, с которыми может работать Copilot, таких как рабочее пространство в целом, терминал и VS Code. Используя участника в запросе, вы с помощью промпта даете Copilot указание задействовать эту область как источник контекста.

В одном из предыдущих подразделов мы спросили Copilot: «Где в этом коде используются импортированные элементы?» (см. рис. 7.7) Copilot вернул общий ответ с описанием последовательности действий для нахождения импортированных элементов в любом проекте Go — вместо нашего проекта, который нас интересует. То же самое случилось, когда мы ввели промпт «Как мне протестировать этот код Go?», — Copilot выдал общие рекомендации о том, как получить эту информацию.

Но если мы добавим в промпт `@workspace`, то Copilot предоставит информацию из файлов, являющихся частью этого рабочего пространства (рис. 7.8). Промпт выглядит так: «@workspace, где в этом коде используются импортированные элементы?» Результат соответствует исходному замыслу.



@workspace Where are imports used in this code?

Workspace

> Used 16 references

✓ Collecting workspace information

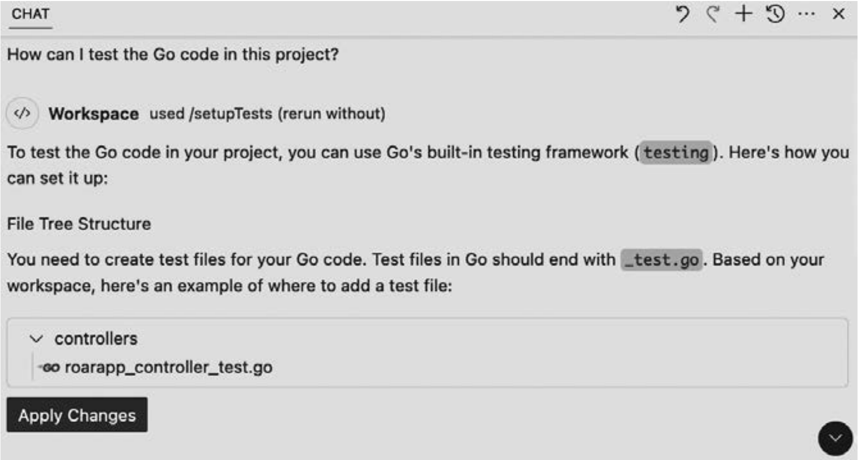
Imports are used throughout the codebase to include external libraries and internal modules. Here are some examples of where and how imports are used in the provided excerpts:

1. **In** roarapp_controller.go :
   - External libraries like context , `logr`, and Kubernetes-related packages (e.g., corev1 , `metav1`, labels) are imported and used in the Reconcile function for managing Kubernetes resources.
   - Internal modules like roarappv1alpha1 are imported to work with custom resource definitions (CRDs).
2. **In** main.go :

Рис. 7.8. Контекст с @workspace

Кроме того, можно неявно подтолкнуть Copilot к использованию `@workspace`. Определенные ключевые слова или фразы в запросе заставляют Copilot добавить `@workspace` в запрос. Например, промпт о тестировании можно изменить так: «Как мне протестировать код Go в этом проекте?» Ссылки на *проект* достаточно, чтобы Copilot понял, что мы подразумеваем более широкий контекст.

В результате он автоматически выполняет запрос, используя `@workspace` и слеш-команду `/setupTests` (рис. 7.9).



How can I test the Go code in this project?

Workspace used /setupTests (rerun without)

To test the Go code in your project, you can use Go's built-in testing framework (`testing`). Here's how you can set it up:

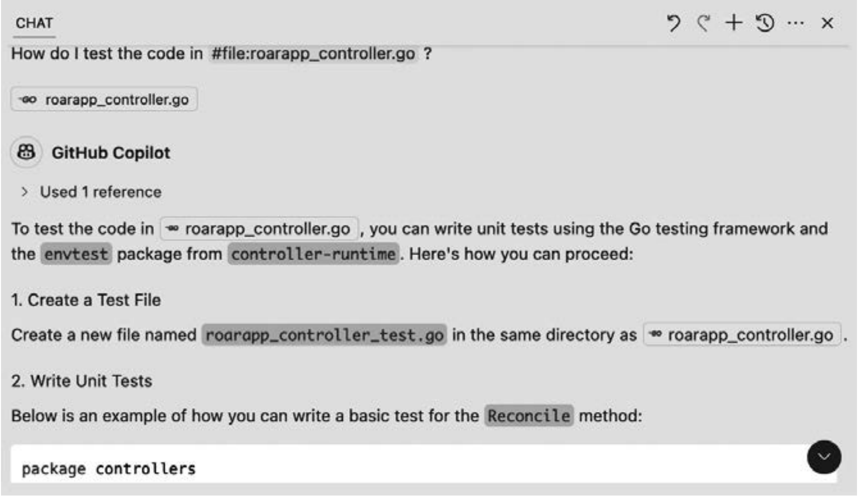
File Tree Structure

You need to create test files for your Go code. Test files in Go should end with `_test.go`. Based on your workspace, here's an example of where to add a test file:

```
controllers
  roarapp_controller_test.go
```

Рис. 7.9. Неявное использование @workspace на основании текста промпта

Если вы хотите сузить контент, то добавьте в промпт одну из чат-переменных для определения контекста, например `#file` или `#selection`. Используя чат-переменную `#file`, можно спросить Copilot, как тестировать код из файла нашего проекта (рис. 7.10).



How do I test the code in #file:roarapp_controller.go ?

roarapp_controller.go

GitHub Copilot

Used 1 reference

To test the code in roarapp_controller.go, you can write unit tests using the Go testing framework and the `envtest` package from `controller-runtime`. Here's how you can proceed:

1. Create a Test File

Create a new file named `roarapp_controller_test.go` in the same directory as roarapp_controller.go.

2. Write Unit Tests

Below is an example of how you can write a basic test for the `Reconcile` method:

```
package controllers
```

Рис. 7.10. Тестирование кода Go с помощью чат-переменной #file

Определение чат-переменных и аргументов

Простой ввод чат-переменных и аргументов напрямую не всегда приводит к получению хороших результатов. Но есть прием, который работает всегда: начните вводить чат-переменную (#), а затем в раскрывающемся списке выберите нужный вариант; при необходимости используйте клавиши со стрелками, чтобы перемещаться между вариантами, и клавишу Enter для выбора нужной переменной. Всегда выбирайте все нужные файлы из списка.

Обратите внимание: при использовании чат-переменной `#file` файл не обязательно должен быть открыт или активен в редакторе. Этот механизм позволяет указать Copilot содержимое проекта, релевантное для получения контекста, и переопределяет стандартные механизмы Copilot для определения нужного вам контента.

Даже если контекст задан явно, Copilot в зависимости от времени обучения модели все равно может выдать устаревший код или неактуальную информацию. Или возвращенные им ответы могут казаться недостаточно релевантными или полными. Но Copilot предоставляет несколько моделей на выбор, поэтому, возможно, есть простой способ решить эту проблему.

Смена модели

Copilot позволяет выбрать одну из ИИ-моделей для получения ответов в чате и предложений кода. У каждой модели есть свои достоинства и недостатки. Например, некоторые модели лучше справляются с генерированием кода. Кроме того, разные модели обучались в разные периоды времени, поэтому новые модели будут более актуальными и сгенерируют устаревший код с меньшей вероятностью, чем старые.

Напомню, что сменить модель можно в чат-интерфейсе. Достаточно выбрать имя используемой модели в раскрывающемся списке (рис. 7.11).

Ранее в этой главе я приводил пример, в котором Copilot генерирует функцию для получения случайных чисел с использованием устаревшего метода (см. рис. 7.1). Эта функция была сгенерирована моделью GPT-4o.

Если вместо этого переключиться на более новую модель (скажем, Gemini 2.5 Pro) и ввести тот же промпт, то Copilot создаст код, используя более новую стандартную функцию `rand.NewSource`, вместо того чтобы пытаться использовать устаревшую функцию `rand.Seed`. Кроме того, если в главной области чата задать вопрос о том, является ли эта функция устаревшей, то Copilot правильно ответит, что, начиная с версии Go 1.20, функция `rand.Seed` считается устаревшей. На рис. 7.12 показан ответ при использовании более новой модели.

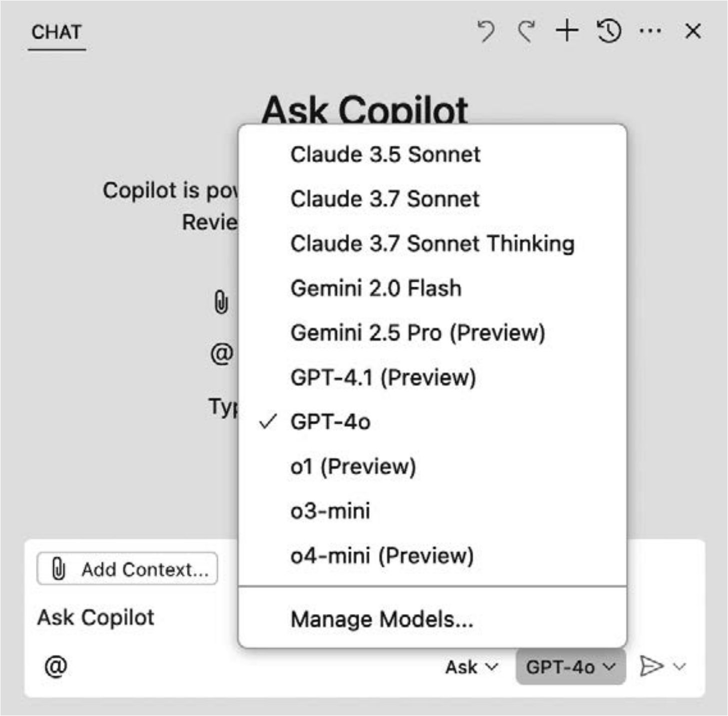
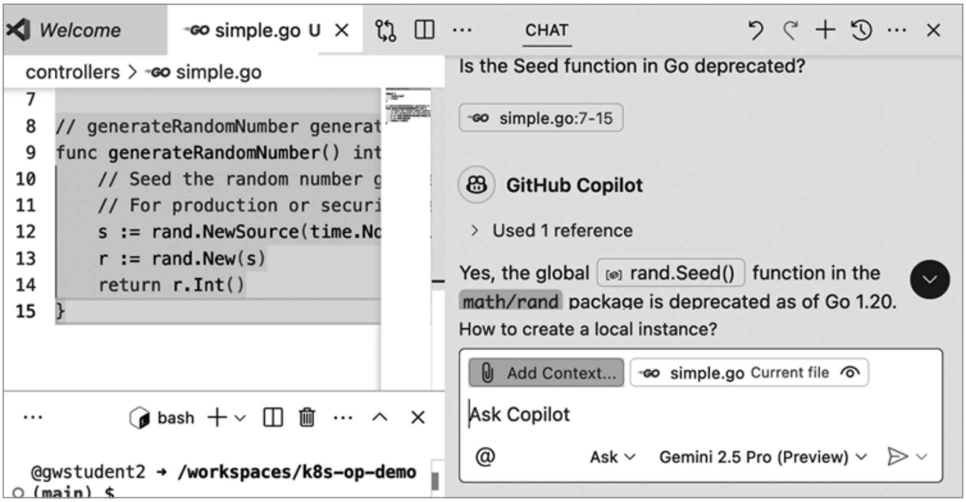


Рис. 7.11. Смена модели в чат-интерфейсе



```
// generateRandomNumber generat
func generateRandomNumber() int
    // Seed the random number g
    // For production or securi
    s := rand.NewSource(time.Nc
    r := rand.New(s)
    return r.Int()
}
```

```
@gwstudent2 ➜ /workspaces/k8s-op-demo
(main) $
```

Is the Seed function in Go deprecated?

Yes, the global rand.Seed() function in the math/rand package is deprecated as of Go 1.20.

How to create a local instance?

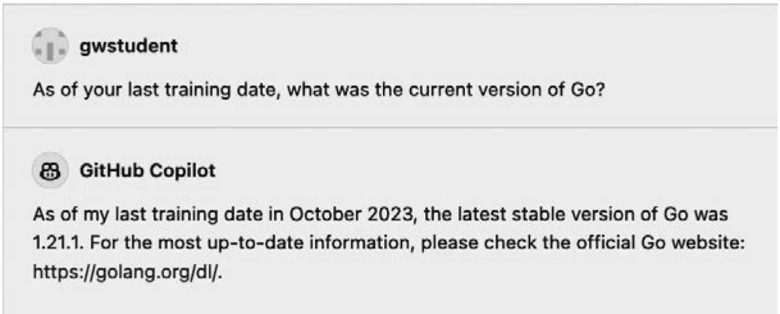
Рис. 7.12. При использовании более новой модели Copilot выдает правильный код и ответ

Но если выбрать другую модель, то как узнать, будет ли она более актуальной? Обычно список обновляется по мере появления новых моделей — как правило, в течение какого-то времени они помечаются как предназначенные для предварительного ознакомления *(preview)*. Чтобы узнать, есть ли более новые версии, можно выполнить поиск в Интернете или на таких сайтах, как Hugging Face (http://huggingface.co).

Кроме того, информацию о том, какая версия чего-либо была текущей на момент обучения модели, можно получить, если обратиться с запросом к самой модели через Copilot.

Запрос к модели о текущей версии

Продолжая пример, в котором Copilot генерировал устаревший код, посмотрим, какую еще информацию мы можем найти. Copilot не понимал, что функция была устаревшей, поэтому нам будет полезно знать, какую версию Go он считает текущей. Можно задать в чат-интерфейсе простой вопрос: «Какая версия Go была текущей на момент твоего последнего обучения?» (рис. 7.13).



gwstudent

As of your last training date, what was the current version of Go?

GitHub Copilot

As of my last training date in October 2023, the latest stable version of Go was 1.21.1. For the most up-to-date information, please check the official Go website: https://golang.org/dl/.

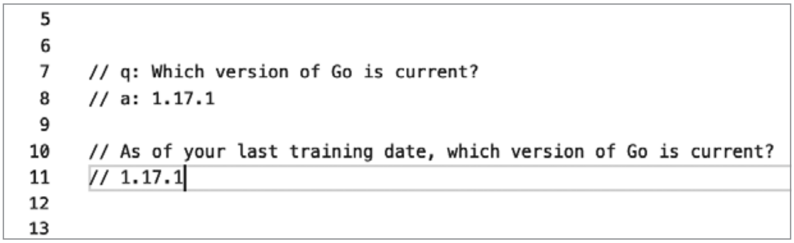
Рис. 7.13. Вопрос к Copilot о текущей версии Go

Правильная формулировка вопроса

В вопросе, заданном Copilot, использовалась конструкция «На момент твоего последнего обучения...». Это было сделано намеренно. Если просто спросить Copilot: «Какая версия Go является текущей?» — то, скорее всего, он расскажет вам, как самостоятельно найти информацию о локально установленном экземпляре Go.

Модель, активная в Copilot, отвечает, что ее последнее обучающее обновление произошло в октябре 2023 года — и на эту дату последней стабильной версией Go была версия 1.21.1.

Возникает очевидный вопрос: если Copilot знает, что новейшей версией была версия 1.21.1, то почему не знает о том, что в версии 1.20 что-то устарело? Вспомните: мы спрашивали Copilot, какая версия была текущей на момент последнего обучения модели, а не какая версия использовалась тогда в репозиториях. Более того, если вы несколько раз зададите Copilot аналогичный вопрос через комментарии в редакторе, то, скорее всего, получите другой ответ (рис. 7.14). Дело в том, что ответ, возвращаемый в редакторе, базируется на различиях в версиях, имеющихся в обучающих данных.



```


// q: Which version of Go is current?
// a: 1.17.1

// As of your last training date, which version of Go is current?
// 1.17.1


```

Рис. 7.14. Вопрос о текущей версии Go, заданный в редакторе

Итак, если считать, что по запросу в чате возвращается информация о самой последней версии, о которой знает модель, но сама она обучалась на других версиях, то что это нам дает?

Информация, возвращаемая в чате, указывает на последнюю версию, которая могла быть представлена в обучающих данных модели. Таким образом, любой код, сгенерированный строго на основании ее обучения, не будет содержать никаких устаревших элементов или новых функций и изменений, произошедших с момента выхода этой версии.

Результаты могут различаться

Не все модели возвращают четкий ответ на вопрос о последней версии. Некоторые могут просто ответить, что эта информация им недоступна, или предложить поискать информацию о последней версии.

Вы можете расширить функционал Copilot, добавив возможности поиска информации о новейших версиях языков и платформ. В главе 10 вы узнаете, как реализовать пример расширения для нахождения новейшей версии Go.

Осознав этот факт, вы сможете помочь Copilot узнать об обновлениях, используя переключение на другую модель или метод, который мы рассмотрим в следующем подразделе.

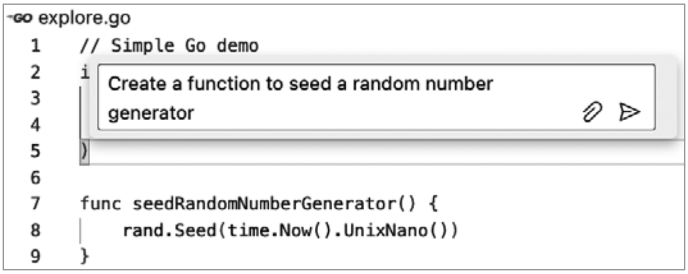
Запросы уровня премиум

Смена модели может быть хорошим способом решения части проблем. Но помните, что использование некоторых расширенных моделей рассматривается как *запросы уровня премиум*. На некоторых тарифных планах использование этих моделей может привести к ускоренному расходованию квот и дополнительным затратам.

А что делать, если ИИ создает устаревший или неполный код из-за того, что вы пользуетесь конкретной моделью, или потому, что модели, которые вы используете, не прошли обучение на данных с информацией о новых функциях? Можно применить еще один подход, чтобы Copilot сгенерировал нужный вам код, используя необходимый контекст.

Обучение Copilot на примере

Вернемся к примеру с устаревшим кодом: Copilot создал функцию для инициализации генератора случайных чисел, используя устаревший метод (рис. 7.15).



```
// Simple Go demo
i



)

func seedRandomNumberGenerator() {
    rand.Seed(time.Now().UnixNano())
}
```

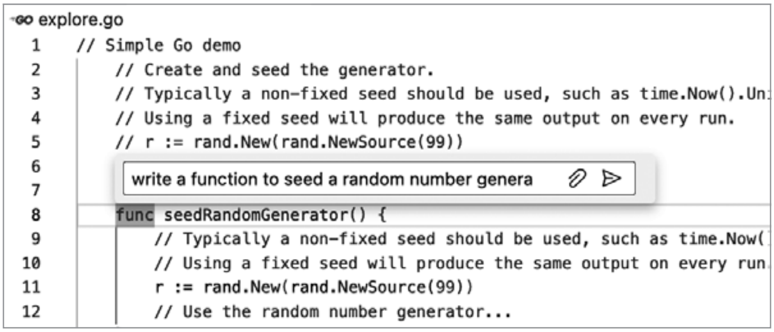
Create a function to seed a random number generator

Рис. 7.15. Функция, созданная с использованием устаревшего метода

Если вы хотите, чтобы Copilot генерировал код, используя замену для устаревших конструкций, то обучите его на примере. Можно взять обновленный фрагмент кода из документации Go (https://oreil.ly/4vDIl) и временно вставить его в тот же файл в вашем рабочем пространстве (или вставить в промпт, если вы используете чат-интерфейс).

Если после этого повторить промпт, то Copilot сгенерирует обновленный код, основанный на добавленном фрагменте кода (рис. 7.16).

Разумеется, это очень простой пример. В данном случае Copilot фактически скопировал предложенный код. И, несмотря на это, конечный результат оказался именно таким, который был нужен. Получив желаемый результат, код примера можно удалить.



```
// Simple Go demo
    // Create and seed the generator.
    // Typically a non-fixed seed should be used, such as time.Now().Un
    // Using a fixed seed will produce the same output on every run.
    // r := rand.New(rand.NewSource(99))

  write a function to seed a random number genera

    func seedRandomGenerator() {
        // Typically a non-fixed seed should be used, such as time.Now(
        // Using a fixed seed will produce the same output on every run
        r := rand.New(rand.NewSource(99))
        // Use the random number generator...
```

Рис. 7.16. Обновленный код, сгенерированный с использованием примера кода

Получение доступа к обновленному коду

Чтобы Copilot мог использовать отдельный контент, в одних ситуациях достаточно поместить его в другой файл, открытый в вашей IDE. В других его нужно будет поместить непосредственно в файл, над которым вы работаете. В таком случае можно добавить контент, а после генерации правильного кода удалить его.

В примерах, приведенных в этом разделе, были показаны способы решения проблем с контентом, уже созданным Copilot. Но вы можете применить аналогичные стратегии еще до генерации, чтобы сразу получить лучшие результаты. Этой теме будет посвящен последний раздел текущей главы.

Добавление контекста для повышения релевантности кода

Помимо компенсации устаревших функций, дополнительный контент, который вы предлагаете Copilot, может помочь в решении еще одной задачи. Когда вы предоставляете более подробный контекст в виде определений или примеров кода, относящихся к вашему проекту, Copilot может использовать их для создания более подробных, содержательных и релевантных рекомендаций по программированию.

В качестве примера рассмотрим создание контента SQL-запросов с помощью Copilot. Допустим, мы разрабатываем систему для управления данными об университетских курсах, студентах, преподавателях и т. д. (мы говорили о ней в главе 5). Не добавляя дополнительный контекст, попросим Copilot создать общий оператор `SELECT` для получения данных о студентах, зачисленных на учебный курс. Для этого можно просто добавить в редакторе такой SQL-комментарий: `-- определи команду select для получения данных обо всех студентах, зачисленных на курс`.

Получив эту общую инструкцию без дополнительного контекста, Copilot выдаст логичный общий запрос:

```
SELECT * FROM students WHERE course_id = 1;
```

В нем нет никаких ошибок. Он абсолютно корректный, учитывая ограниченный контекст, который мы предоставили Copilot. Если предположить, что в нашей базе данных уже есть соответствующие имена полей и таблица, то этот запрос будет работать нормально.

А теперь представим, что у нас уже есть обширный набор баз данных (с соответствующими таблицами, данными и т. д.) для нашей системы. Мы хотели бы, чтобы Copilot сгенерировал предложения, релевантные этим данным, и использовал элементы, которые мы уже определили. Как этого добиться?

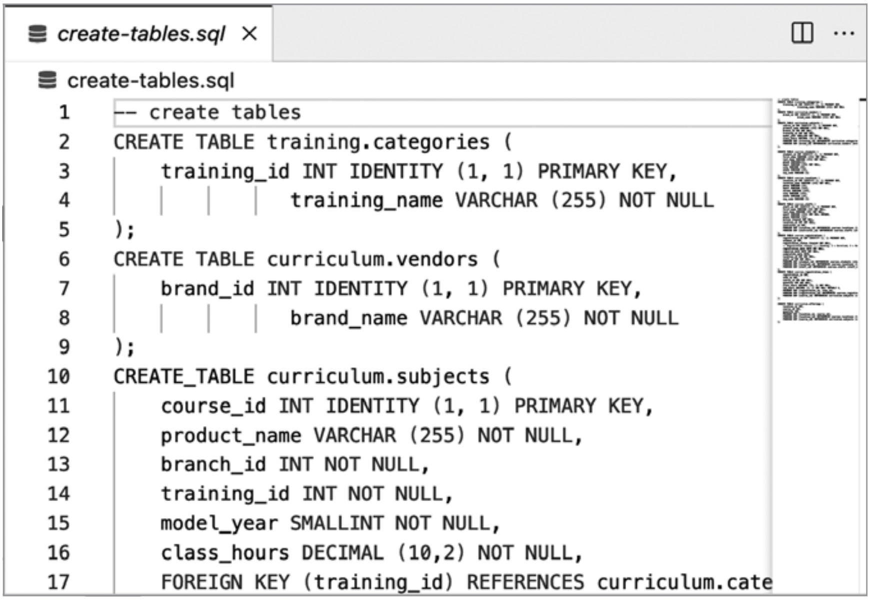
Механизм реализации аналогичен тому, который мы использовали в примере с устаревшим элементом (см. предыдущий раздел): нужно добавить больше контекста, который Copilot сможет извлечь из рабочего пространства. В данном случае, хотя у нас нет возможности попросить Copilot динамически проанализировать хранилища данных, индексы или хранимые процедуры, мы можем предоставить их статическое представление в виде определений схем баз данных. Для этого можно либо использовать исходное определение SQL, если оно доступно, либо сгенерировать схемы по существующим данным. Часть файла, который может использоваться для этого, показан на рис. 7.17.

Файл содержит таблицы, определенные для областей, с которыми мы работаем, а также необходимые поля, ключи и т. д.

Добавим этот файл в рабочее пространство, чтобы он стал частью проекта, и откроем его в редакторе. Теперь Copilot может извлечь дополнительный контекст. Вернемся назад и предоставим ту же инструкцию в форме комментария: `-- определи команду select для получения данных обо всех студентах, зачисленных на курс`. На этот раз Copilot выдает следующий ответ:

```
SELECT students.first_name, students.last_name, students.email,
 students.phone, students.city, students.state, students.zip_code
FROM courses.students
JOIN courses.registrations
ON students.student_id = registrations.student_id
JOIN courses.registration_items
ON registrations.registration_id =
registration_items.registration_id
WHERE registration_items.course_id = 1;
```

Теперь, при наличии дополнительного контекста из другого файла, доступного в рабочем пространстве, Copilot выдал намного более подробный и релевантный запрос. Вдобавок он использовал конкретную таблицу и имена полей из другого файла.



```
-- create tables
CREATE TABLE training.categories (
    training_id INT IDENTITY (1, 1) PRIMARY KEY,
               training_name VARCHAR (255) NOT NULL
);
CREATE TABLE curriculum.vendors (
    brand_id INT IDENTITY (1, 1) PRIMARY KEY,
               brand_name VARCHAR (255) NOT NULL
);
CREATE_TABLE curriculum.subjects (
    course_id INT IDENTITY (1, 1) PRIMARY KEY,
    product_name VARCHAR (255) NOT NULL,
    branch_id INT NOT NULL,
    training_id INT NOT NULL,
    model_year SMALLINT NOT NULL,
    class_hours DECIMAL (10,2) NOT NULL,
    FOREIGN KEY (training_id) REFERENCES curriculum.cate
```

Рис. 7.17. Расширенное определение схемы, которое будет использоваться в качестве контекста

Важно, что Copilot, принимая решения о том, как генерировать код, отдает приоритет контексту, который используется в IDE. Это полезная и простая стратегия, позволяющая получать от Copilot гораздо более практичные и актуальные предложения кода и ответы, чем если бы вы использовали только данные для обучения модели.

Добавление дополнительных файлов в качестве контекста

В рассмотренном выше примере с SQL мы добавили дополнительный файл в рабочее пространство и открыли его в IDE. Мы не добавляли его содержимое в активный файл, как делали в примере с устаревшим элементом. Этот способ тоже был возможен, но использовать его здесь не обязательно.

В зависимости от ситуации вы можете обнаружить, что один из этих двух способов (вставка контекста непосредственно в активный файл или открытие контента в отдельном файле) подходит вам лучше. Попробуйте всегда начинать со второго способа. Если при этом вы не получите ожидаемый результат, то добавьте контент в активный файл.

Другой возможный вариант — вставка релевантных примеров прямо в Copilot Chat (если масштаб примера это позволяет).

Если вы добавляете контент в IDE (будь то новый файл или дополнение к существующему), обязательно сохраните изменения перед отправкой запроса или промпта. Контент, сохраненный в рабочем пространстве, Copilot использует в качестве контекста.

Суть в том, что предварительное добавление расширенного контекста для Copilot может привести к улучшенным предложениям по коду или вариантам завершения кода. В некоторых случаях генерирование файлов (таких как схема в примере выше) может занимать больше времени, но оно компенсируется за счет снижения количества времени и усилий, затрачиваемых на доработку предложений Copilot, необходимую для достижения нужной детализации.

РЕКОМЕНДАЦИИ ПО УПРАВЛЕНИЮ COPILOT

Вот еще несколько советов по поводу того, как помочь Copilot, если вы не получили желаемый контент, а другие стратегии не дают результата.

- Переключайтесь со встроенных предложений на промпты в чате (или наоборот), если полученные результаты вас не устраивают.
- Если Copilot раз за разом генерирует неправильный код, то попробуйте написать его часть (заглушку) и попросите сгенерировать завершение — возможно, итоговый результат будет лучше.
- Добавляйте более конкретный комментарий, в котором четко укажите, что вам нужно.
- Временно поместите код в отдельное окно, чтобы исключить другой контекст, который может искажать результаты.
- Просите Copilot объяснить код, а не генерировать его.
- Изменяйте имя (и даже тип) файла, чтобы обновить контекст.

Резюме

В этой главе мы рассмотрели, как Copilot определяет контекст для своих предложений и ответов по коду, обсудили возможные причины неточных или неактуальных ответов, а также способы решения подобных проблем.

Основной контекст Copilot формируется из обучения используемых им базовых моделей. Все они актуальны лишь в тот момент времени, когда проходили последнее обучение, в них не учитываются обновления или изменения в языках или фреймворках, которые не использовались при обучении. Кроме того, в контенте, на котором обучались модели, могли использоваться старые версии инструментария. Поэтому предложения или ответы, генерируемые такими моделями, могут содержать устаревшую информацию. Это ключевой момент, о котором нужно помнить и который необходимо постоянно отслеживать. Одно из возможных простых решений — переключение на обновленную модель (если она доступна).

К счастью, Copilot извлекает контекст не только из моделей, но и из доступного контента в редакторе и используемом рабочем пространстве. Этот контекст обладает более высоким приоритетом, чем обучающие данные, поскольку отражает то, чем вы занимаетесь в настоящий момент. Используя этот подход, вы предоставляете Copilot более актуальный и более релевантный контекст. Для этого собственный релевантный контент вы должны добавить в IDE и рабочее пространство. Во многих случаях для этого файл с соответствующей информацией достаточно открыть в среде. Чтобы передать более явные указания, релевантную информацию, которую должен использовать Copilot, можно добавить в текущий файл время, необходимое Copilot для работы с ней.

Кроме того, GitHub Copilot поддерживает функции, позволяющие сосредоточить внимание ИИ на определенном контенте. Участники чата могут определять общие контекстные области, в рамках которых Copilot будет отвечать на вопросы. На момент написания книги поддерживаются участники для VS Code, активного терминала и вашего рабочего пространства.

Если вы хотите, чтобы Copilot при работе с контекстом сосредоточился на конкретных элементах вашего рабочего пространства, то используйте чат-переменные. Примеры таких элементов — текущее выделение в редакторе, открытый в редакторе контент, последняя команда в терминале и т. д.

И наконец, вы узнали, как предоставить GitHub Copilot больше контекста при работе с SQL. Это пример нестандартного языка/фреймворка, при взаимодействии с которым тоже можно использовать Copilot. В следующей главе мы рассмотрим примеры применения Copilot в нетипичных областях.

ГЛАВА 8

Другие способы использования Copilot

Как вы видели, Copilot очень полезен в традиционных рабочих процессах программирования, а также помогает в тестировании и документировании. Он может помочь и во многих менее распространенных сценариях, с которыми сталкиваются программисты.

Мы не будем рассматривать все эти сценарии в этой главе (и даже в книге), но обсудим несколько характерных примеров. Для начала стоит сказать, что структура этой главы отличается от структуры других глав. Мы исследуем множество разнообразных сценариев, вместо того чтобы концентрироваться на одной общей функции или функциональной области. Примеры, с которыми мы будем работать, кратко описаны ниже.

Возможно, первое, что приходит в голову, — использование GitHub Copilot для программирования на популярных языках (таких как JavaScript и Python). Но Copilot имеет доступ ко всем языкам, на которых обучалась используемая вами модель. Поэтому вы можете использовать его и для тех языков программирования, которые не считаются традиционными. Пример — язык SQL (см. предыдущие главы), часто используемый для работы с реляционными базами данных. Copilot может упростить огромное количество операций, связанных с базами, когда вы работаете с запросами, схемами и хранимыми процедурами.

Фреймворки — такие как Kubernetes — используют упорядоченный формат (чаще всего выражаемый разметкой YAML). Они относятся к категории декларативных, а не императивных, но их синтаксис и структура понятны Copilot. Как и для любого языка программирования, он может предоставить рекомендации по завершению кода и ответы о том, как работать с ними.

Наконец, есть регулярные выражения — строки символов, часто сложные и непонятные. Правильно написать их важно, но обычно очень непросто. Copilot может упростить работу с такими выражениями.

Для тестирования регулярных выражений или юнит-тестирования функций вам могут понадобиться структурированные тестовые данные. При наличии специальных промптов Copilot может генерировать практически любые необходимые структуры данных, хотя есть несколько нюансов, которые нужно учитывать.

Наконец, стоит заметить, что часть функциональности Copilot внедрена в CLI-приложение GitHub. В настоящее время его возможности ограничены, но можно ожидать, что со временем будет добавлена новая функциональность.

В этой главе мы рассмотрим все эти темы. Начнем с использования Copilot при работе с SQL.

Использование Copilot с SQL

SQL формально является языком программирования, но применяет специфические конструкции для управления данными: стандартные запросы, используемые с данными и схемами баз данных, индексы для ускорения поиска и хранимые процедуры для решения более сложных задач.

В этом разделе мы рассмотрим три области применения Copilot:

- запросы;
- хранимые процедуры;
- оптимизации.

Начнем с основных способов использования Copilot для генерирования запросов, связанных с SQL.

Запросы

Как вы знаете, Copilot извлекает информацию из локального контента, чтобы получить контекст. Поэтому, если вы работаете в редакторе и открываете файл с расширением `.sql` или содержащий команды SQL, Copilot может определить это и выбрать то, что необходимо для генерирования SQL. Кроме того, вы можете использовать комментарии, чтобы задать работе Copilot нужное вам направление.

В главе 7 приводился пример того, как добавить контекст, чтобы повысить релевантность генерирования кода и предложений. Там мы использовали прием открытия файла SQL со всеми определениями схем университетской базы данных, чтобы предоставить Copilot больше контекста для генерирования запросов.

Если вы успели забыть, о чем идет речь, то вернитесь к разделу «Добавление контекста для повышения релевантности кода» главы 7. Здесь будет предполагаться та же конфигурация. Частичное определение схемы, лежащей в основе системы регистрации, выглядит так:

```
-- создай таблицы
...

CREATE TABLE courses.students (
    student_id INT IDENTITY (1, 1) PRIMARY KEY,
    first_name VARCHAR (255) NOT NULL,
    last_name VARCHAR (255) NOT NULL,
    phone VARCHAR(25),
    email VARCHAR (255) NOT NULL,
    city VARCHAR (50),
    state VARCHAR (25),
    zip_code VARCHAR (5)
);

...

CREATE TABLE courses.registrations (
    registration_id INT IDENTITY (1, 1) PRIMARY KEY,
    student_id INT,
    registration_status tinyint NOT NULL,
    -- Registration status: 1 = Pending; 2 = Enrolled; 3 = Rejected;
 4 = Completed
    registration_date DATE NOT NULL,
    required_date DATE NOT NULL,
    completion_date DATE,
    location_id INT NOT NULL,
    staff_id INT NOT NULL,
    FOREIGN KEY (student_id) REFERENCES courses.students(student_id)
 ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (location_id) REFERENCES courses.locations
    (location_id) ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (staff_id) REFERENCES courses.staffs (staff_id) ON
    DELETE NO ACTION ON UPDATE NO ACTION

...

CREATE TABLE courses.registration_items (
    registration_id INT,
    item_id INT,
    course_id INT NOT NULL,
    quantity INT NOT NULL,
    class_hours DECIMAL (10, 2) NOT NULL,
    lab_hours DECIMAL (4, 2) NOT NULL DEFAULT 0,
    PRIMARY KEY (registration_id, item_id),
    FOREIGN KEY (registration_id) REFERENCES courses.registrations
    (registrations_id) ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (course_id) REFERENCES curriculum.subjects
    (course_id) ON DELETE CASCADE ON UPDATE CASCADE
);
```

Если вы откроете это определение в редакторе, в котором работаете над запросами, а затем, используя комментарий, предложите Copilot сгенерировать запрос SQL для получения данных обо всех студентах, зачисленных на курс, то получите подробный результат с именами таблиц, полей и т. д., соответствующих схеме:

```
-- определи команду select для получения данных обо всех студентах,
зачисленных на курс

SELECT students.first_name, students.last_name, students.email,
  students.phone, students.city, students.state, students.zip_code
FROM courses.students
JOIN courses.registrations
ON students.student_id = registrations.student_id
JOIN courses.registration_items
ON registrations.registration_id =
registration_items.registration_id
WHERE registration_items.course_id = 1;
```

Просто из-за того, что в рабочем пространстве был открыт другой файл, Copilot смог использовать этот контент для создания улучшенного запроса.

Сохраните работу

Не забудьте сначала сохранить любые новые файлы как часть проекта, если хотите, чтобы Copilot мог использовать их в контексте.

Другой пример: можно попросить Copilot создать таблицу посещаемости занятий студентами. Учитывая тот же контекст более крупной таблицы со схемой, он должен выдать полное определение таблицы:

```
-- определи таблицу посещаемости студентов по занятиям
CREATE TABLE courses.attendance (
    attendance_id INT IDENTITY (1, 1) PRIMARY KEY,
    registration_id INT NOT NULL,
    attendance_date DATE NOT NULL,
    attendance_status tinyint NOT NULL,
    -- Статус посещения: 1 = присутствует; 2 = отсутствует; 3 = опоздание;
       4 = отсутствует по уважительной причине
    FOREIGN KEY (registration_id) REFERENCES courses.registrations
    (registration_id) ON DELETE CASCADE ON UPDATE CASCADE
);
```

Формат этой таблицы напоминает формат, использованный в файле схемы для таблицы `courses.registration`, как показано в листинге ниже. Обратите внимание: в обоих определениях таблицы используются значения статуса 1–4:

```
CREATE TABLE courses.registrations (
    registration_id INT IDENTITY (1, 1) PRIMARY KEY,
    student_id INT,
```

```
    registration_status tinyint NOT NULL,
    -- Статус регистрации: 1 = не завершена; 2 = зачислен; 3 = отказ;
       4 = завершен
    registration_date DATE NOT NULL,
    required_date DATE NOT NULL,
    completion_date DATE,
    location_id INT NOT NULL,
    staff_id INT NOT NULL,
    FOREIGN KEY (student_id) REFERENCES courses.students
    (student_id) ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (location_id) REFERENCES courses.locations
    (location_id) ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (staff_id) REFERENCES courses.staffs (staff_id)
     ON DELETE NO ACTION ON UPDATE NO ACTION
```

Кроме того, можно попросить Copilot генерировать более сложные типы данных SQL — например, хранимые процедуры.

Хранимые процедуры

Хранимая процедура — готовый код SQL, который хранится в базе данных и может легко использоваться повторно. Copilot помогает генерировать такие процедуры. Продолжая пример с университетской системой регистрации, попросите Copilot создать новую хранимую процедуру для получения списка студентов, зачисленных по месту проживания. Чтобы сообщить Copilot, что нужно сделать, можно использовать простой промпт в чате (по аналогии с формой комментариев, использованной ранее): «Определи хранимую процедуру для получения информации о зачислении на курсы по месту проживания».

Используя этот промпт и дополнительный контекст схем, Copilot выдает логичный результат:

```
CREATE PROCEDURE GetCourseEnrollmentByLocation
    @CourseID INT
AS
BEGIN
    SELECT students.city, students.state, COUNT(*) AS
enrollment_count
    FROM courses.students
    JOIN courses.registrations
    ON students.student_id = registrations.student_id
    JOIN courses.registration_items
    ON registrations.registration_id =
registration_items.registration_id
    WHERE registration_items.course_id = @CourseID
    GROUP BY students.city, students.state;
END
```

Можно дать и более четкие инструкции. Чем подробнее будет запрос, тем скорее Copilot сгенерирует лучшие результаты. Например, возьмем такой конкретный промпт в чат-интерфейсе:

```
Определи хранимую процедуру для получения подробной информации
о преподавателях, связанной с их местами работы, добавь подробную
информацию о преподавателе, подробную информацию о месте работы
и учебных курсах, связанную с преподавателем. На входе процедура
получает instructor_id
```

Copilot сначала объяснил свои рассуждения, изложив их в виде плана, а затем сгенерировал подробный код в соответствии с запросом:

```
План
1. Определить новую хранимую процедуру с именем `GetInstructorDetailsByLocation.
2. Получить входной параметр `@InstructorID`.
3. Выбрать подробную информацию о преподавателе, месте работы и курсах,
   связанную с преподавателем.
4. Использовать соединения для выборки данных
   из `instructors`, `locations` и [`courses`].
5. Сгруппировать результаты по преподавателю и подробной информации
   о месте работы.

Код SQL
CREATE PROCEDURE GetInstructorDetailsByLocation
    @InstructorID INT
AS
BEGIN
    SELECT
        instructors.instructor_id,
        instructors.first_name,
        instructors.last_name,
        instructors.email,
        locations.city,
        locations.state,
        courses.course_id,
        courses.course_name
    FROM
        courses.instructors
    JOIN
        courses.locations
    ON
        instructors.location_id = locations.location_id
    JOIN
        courses.course_instructors
    ON
        instructors.instructor_id = course_instructors.instructor_id
    JOIN
        courses.courses
```

```
    ON
        course_instructors.course_id = courses.course_id
    WHERE
        instructors.instructor_id = @InstructorID;
END;
```

Итак, Copilot генерирует запросы и хранимые процедуры, и это, безусловно, полезно, но работа с SQL не единственный его навык. Например, ИИ может помочь оптимизировать код.

Оптимизации

Copilot может повысить эффективность кода двумя основными способами. Первый — он может написать код, используя встроенные конструкции, предоставляемые языком или фреймворком для оптимизации. Второй — он может оптимизировать существующий (статический) код.

Пример для первого случая: SQL предоставляет индексы для ускорения операций. Обратимся к запросу для получения данных обо всех студентах, зачисленных на курс, с которым мы работали ранее в этой главе. Полный запрос выглядит так:

```
-- определи команду select для получения данных обо всех студентах,
зачисленных на курс

SELECT students.first_name, students.last_name, students.email,
  students.phone, students.city, students.state, students.zip_code
FROM courses.students
JOIN courses.registrations
ON students.student_id = registrations.student_id
JOIN courses.registration_items
ON registrations.registration_id =
registration_items.registration_id
WHERE registration_items.course_id = 1;
```

Для этого запроса можно использовать отдельный индекс для ускорения операций. Мы просим Copilot создать новый индекс на основе запроса, введя промпт «-- напиши индекс для улучшения быстродействия запроса». Для этого запроса Copilot может предложить такую команду SQL:

```
-- напиши индекс для улучшения быстродействия запроса

create index idx_registration_status_date on courses.registrations
  (registration_status, registration_date);
```

Как и в случае с исходным запросом, Copilot смог использовать контекст доступного кода, чтобы создать индекс, в котором используются существующие имена полей, таблиц и т. д.

Помимо создания кода для оптимизации выполнения, Copilot может оптимизировать статический код. Это второй случай, упомянутый в начале этого подраздела. Представьте, что у вас есть запрос для получения данных обо всех регистрациях на учебных курсах на сентябрь 2024 года. Он может выглядеть так:

```
select * from courses.registrations where year(registration_date)
= 2024 and month(registration_date) = 9;
```

Можно выделить запрос, а затем попросить Copilot оптимизировать его, используя промпт «Оптимизируй этот запрос».

ИИ предлагает оптимизацию, которая использует даты начала и окончания регистрации, заданные напрямую:

```
SELECT * FROM courses.registrations WHERE registration_date >=
'2023-09-01' AND registration_date < '2023-10-01'
```

В том же выводе Copilot предоставляет дополнительную информацию о том, *почему* это изменение улучшает быстродействие:

```
…использование функций или столбцов в условии WHERE может помешать
использованию индексов и замедлению запроса. Если registration_date
индексируется, то запрос можно оптимизировать, исключив использование функций
YEAR() и MONTH() и выполнив вместо этого сравнение по диапазону дат...
```

Copilot не только предложил оптимизацию, но и объяснил, почему и при каких условиях это изменение действительно приведет к улучшению. Замечательно!

Итак, вы увидели, как с помощью Copilot можно управлять кодом баз данных. Теперь посмотрим, как он работает с фреймворками, используя декларативные спецификации.

Работа с YAML и Kubernetes

YAML — критически важный язык форматирования, используемый во многих современных приложениях. Работать с ним может быть непросто из-за синтаксиса и жестких требований к форматированию. Но возможность создания спецификаций и конфигураций играет ключевую роль в таких приложениях, как Kubernetes.

Работать с Kubernetes тоже может быть непросто по разным причинам. В частности, необходимо тщательно изучить определение нужных объектов в файлах YAML и запомнить эту информацию. Каждый платформенный объект имеет общий набор атрибутов, как и свою спецификацию. Если вы попытаетесь масштабировать объекты для сервиса или продукта, то проблем может быть еще больше.

Любая ошибка или неправильное форматирование может привести к тому, что ваше приложение не будет работать. Вот почему в этой области можно эффективно применить ИИ.

Допустим, вы хотите запрограммировать новое развертывание в Kubernetes. *Развертывание* (deployment) — разновидность объекта Kubernetes, который создает множественные экземпляры *модулей* (pods) Kubernetes и управляет ими. В свою очередь, модуль содержит один или несколько контейнеров. Они создаются на базе образов и предоставляют автономную среду для выполнения приложения.

Если вы знаете об основных компонентах, то можете в чат-интерфейсе попросить Copilot создать спецификацию YAML для развертывания:

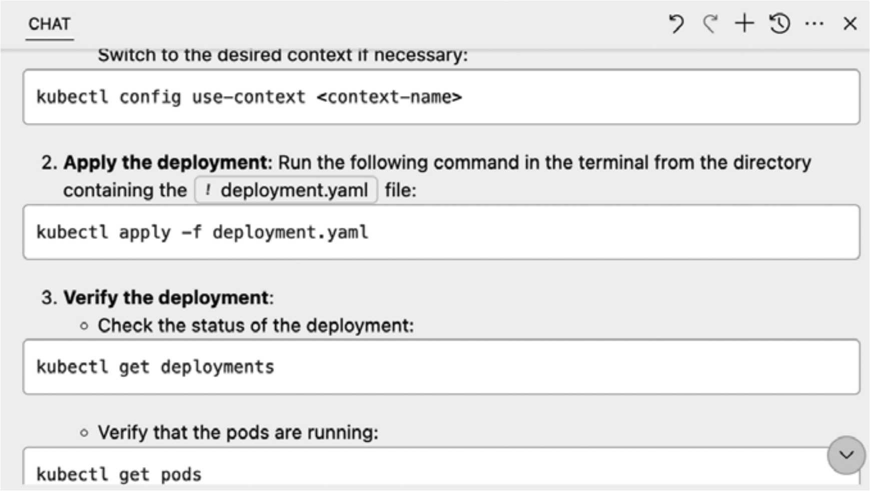
```
Напиши спецификацию для развертывания в Kubernetes с двумя репликами
и образом из busybox
Добавь команду для выполнения в контейнерах: sleep 1600
Добавь метку приложения: myapp
Добавь тип метки: front-end
```

Через пару секунд Copilot предложит такой код:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: busybox-container
        image: busybox
        command: ["sleep", "3600"]
```

На случай если вы не знакомы с Kubernetes, скажу, что это действительный синтаксис, который выглядит именно так, как должна выглядеть спецификация развертывания. Обратите внимание: для генерирования этого кода вам не требовались никакие специальные знания о синтаксисе Kubernetes, кроме типов объектов, которые должны входить в спецификацию. Вы сохраняете код в файле `deployment.yaml` и продолжаете работать.

Имея готовую спецификацию, вы в какой-то момент захотите выполнить ее. Если вам не совсем понятно, как это делается, то спросите Copilot. Просто откройте чат-интерфейс и задайте вопрос: «Как мне это выполнить?» На рис. 8.1 показано, что Copilot обращается к созданному файлу `deployment.yaml` и предоставляет инструкции о том, как реализовать его в кластере Kubernetes (системе машин для выполнения Kubernetes) и убедиться, что развертывание работает. Для этого используется интерфейс командной строки (CLI) Kubernetes, который называется `kubectl`.



Switch to the desired context if necessary:

```
kubectl config use-context <context-name>
```

2. **Apply the deployment:** Run the following command in the terminal from the directory containing the `! deployment.yaml` file:

```
kubectl apply -f deployment.yaml
```

3. **Verify the deployment**:
   - Check the status of the deployment:

```
kubectl get deployments
```

   - Verify that the pods are running:

```
kubectl get pods
```

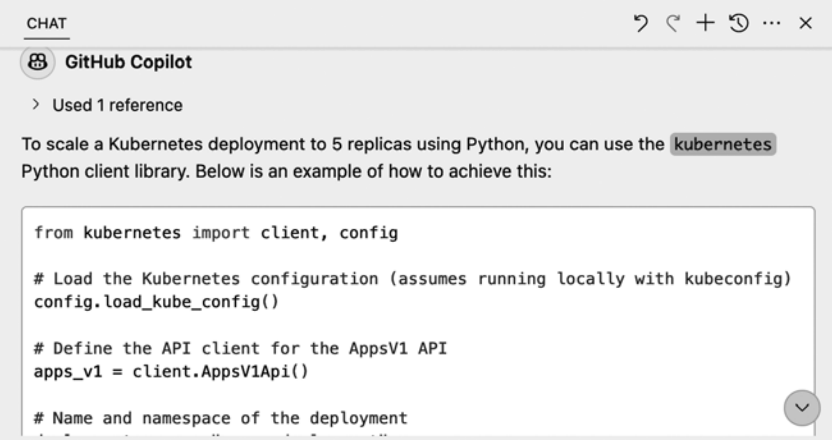
Рис. 8.1. Copilot объясняет, как применить спецификацию Kubernetes для выполнения

Это очень полезно, но что вы будете делать, если в какой-то момент вам понадобится управлять этим развертыванием Kubernetes, используя Kubernetes API вместо CLI? А еще вам хотелось бы сделать это на Python и иметь возможность изменить (масштабировать) количество экземпляров выполняемого контейнера.

Можно спросить об этом у Copilot, введя в чате другой простой промпт на естественном языке:

```
Как мне вызвать K8s API для масштабирования развертывания до 5 реплик в Python?
```

Copilot сначала сообщает, что можно воспользоваться клиентской библиотекой Kubernetes Python, а затем выдает код Python для решения этой задачи (рис. 8.2).

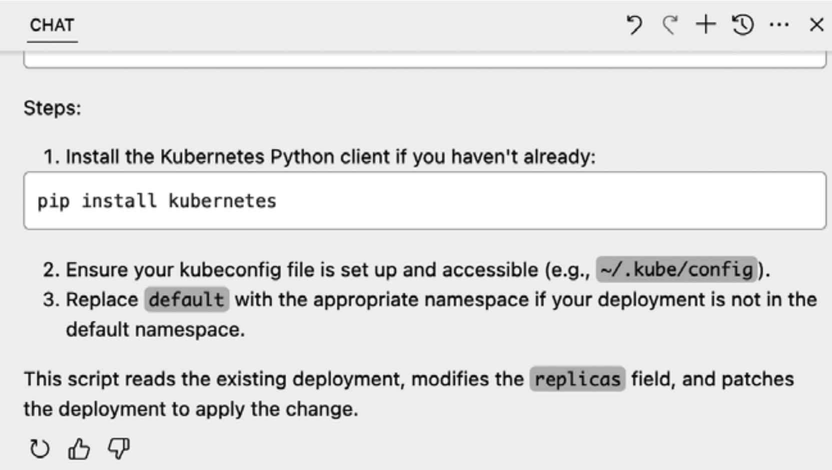


To scale a Kubernetes deployment to 5 replicas using Python, you can use the `kubernetes` Python client library. Below is an example of how to achieve this:

```
from kubernetes import client, config

# Load the Kubernetes configuration (assumes running locally with kubeconfig)
config.load_kube_config()

# Define the API client for the AppsV1 API
apps_v1 = client.AppsV1Api()

# Name and namespace of the deployment
```

Рис. 8.2. Ответ Copilot об управлении Kubernetes в Python

Кроме того, Copilot дает объяснение действий, необходимых для выполнения кода (рис. 8.3).



Steps:

1. Install the Kubernetes Python client if you haven't already:

```
pip install kubernetes
```

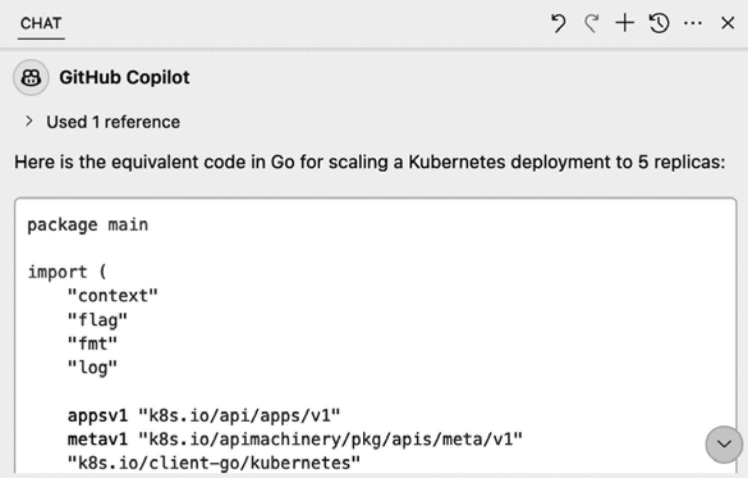
2. Ensure your kubeconfig file is set up and accessible (e.g., `~/.kube/config`).
3. Replace `default` with the appropriate namespace if your deployment is not in the default namespace.

This script reads the existing deployment, modifies the `replicas` field, and patches the deployment to apply the change.

Рис. 8.3. Последовательность выполнения

Предложенный код можно сохранить в новом файле в редакторе и заняться внесением любых необходимых изменений, тестированием кода и т. д.

Рассмотрим еще один пример использования Copilot. Самый популярный язык программирования для Kubernetes — Go. Если вы хотите перенести этот код Python на Go, то может ли Copilot помочь? Оказывается, да! И делается это очень просто. Выделив код Python (который будет использоваться для извлечения контекста), достаточно ввести промпт «Переведи на Go». Copilot генерирует эквивалентные фрагменты кода Go в чат-интерфейсе (рис. 8.4).

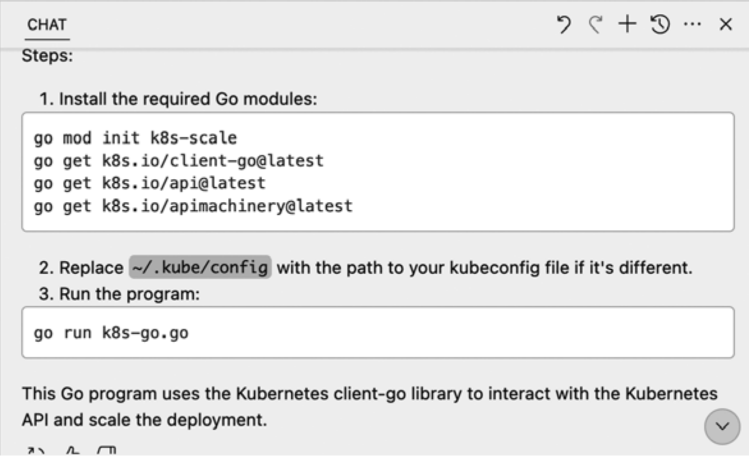


Here is the equivalent code in Go for scaling a Kubernetes deployment to 5 replicas:

```
package main

import (
    "context"
    "flag"
    "fmt"
    "log"

    appsv1 "k8s.io/api/apps/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes"
```

Рис. 8.4. Код развертывания Kubernetes на Go

Кроме того, Copilot предоставляет соответствующее объяснение и инструкции по выполнению, как он делал это для версии на Python (рис. 8.5).



Steps:

1. Install the required Go modules:

```
go mod init k8s-scale
go get k8s.io/client-go@latest
go get k8s.io/api@latest
go get k8s.io/apimachinery@latest
```

2. Replace `~/.kube/config` with the path to your kubeconfig file if it's different.
3. Run the program:

```
go run k8s-go.go
```

This Go program uses the Kubernetes client-go library to interact with the Kubernetes API and scale the deployment.

Рис. 8.5. Последовательность выполнения кода на Go

Это еще один пример того, как Copilot может автоматизировать сложные задачи и упростить их обработку.

Еще одна обычно сложная задача, которую Copilot может значительно упростить, — создание регулярных выражений. Об этом поговорим в следующем разделе.

Генерирование регулярных выражений

Многие разработчики не в восторге от синтаксиса регулярных выражений. Процесс поиска комбинации букв, цифр и знаков, которые можно использовать для эффективного поиска совпадений, может быть долгим и утомительным. Добавьте к этому необходимость писать тесты для комплексной проверки выражений — и станет очевидно, что лучше попробовать найти что-нибудь похожее в другом коде или на таком ресурсе, как Stack Overflow.

Тем не менее это еще одна область, в которой Copilot может быть очень полезным. При наличии подходящего промпта он сможет легко создать соответствующее регулярное выражение. Допустим, вы просите: «Создай регулярное выражение для проверки SSN» (номер социального страхования в США). Copilot выводит в ответ такое предложение реализации и объяснение:

```
Регулярное выражение для проверки номера социального страхования США (SSN):

^\d{3}-\d{2}-\d{4}$

Объяснение:
 • ^ совпадает с началом строки.
 • \d{3} совпадает ровно с 3 цифрами.
 • - совпадает с литералом "дефис".
 • \d{2} совпадает ровно с 2 цифрами.
 • - совпадает еще с одним литералом "дефис".
 • \d{4} совпадает ровно с 4 цифрами.
 • $ совпадает с концом строки.

Пример использования в Python:

python
import re

def validate_ssn(ssn):
    pattern = r"^\d{3}-\d{2}-\d{4}$"
    return bool(re.match(pattern, ssn))

# Тестирование функции
print(validate_ssn("123-45-6789"))  # True
print(validate_ssn("123-456-789"))  # False

Проверяет, что SSN имеет формат `XXX-XX-XXXX`, где `X` - цифра.
```

Возможно, вам также потребуется сгенерировать функцию, которая использует регулярное выражение на том языке, на котором вы работаете. Допустим, у вас есть файл JavaScript (`phone.js`) и вы хотите добавить в него функцию для проверки любого международного телефонного номера.

Введите промпт: «Создай функцию для проверки любого международного телефонного номера по регулярному выражению». Copilot генерирует такой код JavaScript:

```
function validatePhoneNumber(phoneNumber) {
    const phoneRegex = /^\+(?:[0-9] ?){6,14}[0-9]$/;
    return phoneRegex.test(phoneNumber);
}

// Пример использования:
console.log(validatePhoneNumber("+1234567890")); // true
console.log(validatePhoneNumber("1234567890")); // false
```

Код получается коротким, но само регулярное выражение непростое. Кроме того, Copilot предоставил примеры вызовов для проверки функции. Если варианты использования в примерах вам не подходят, то отклоните предложение и введите более подробный промпт, чтобы получить нужный результат.

Когда регулярное выражение будет сгенерировано, вам могут понадобиться простые данные для его тестирования или для базового тестирования другой части вашего кода. Вместо того чтобы искать такие данные или пытаться создать их самостоятельно, вы можете поручить эту задачу Copilot.

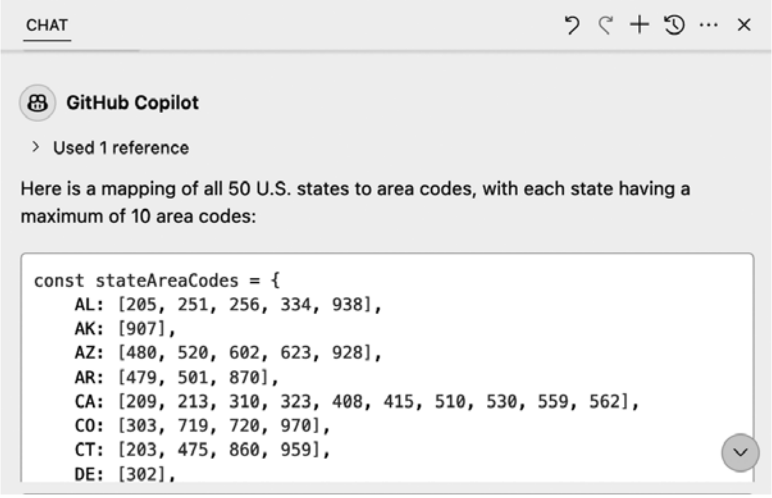
Автоматическое генерирование данных

Еще одна область, в которой Copilot может помогать решать стандартные задачи, — генерирование данных и их сопоставлений для проверки подходов к программированию и выполнения последующего тестирования.

Подход тот же: чтобы получить желаемые результаты, используйте четко определенный промпт. Представьте, что вам нужны тестовые данные для проверки функции, которая ищет телефонные коды регионов для заданного штата. Можно попросить Copilot: «Сопоставь все 50 штатов с телефонными кодами региона; ключом сделай сокращенное обозначение штата, а значением — массив кодов, содержащий не более 10 элементов».

Обратите внимание: мы конкретно определяем ключи и соответствующие значения для сопоставления. Copilot выдает запрашиваемые данные (рис. 8.6).

Иногда Copilot добавляет дисклеймер относительно точности сгенерированных данных. В нашем случае он указывает, что телефонные коды региона — всего лишь подсказка, а их соответствие реальным кодам региона для каждого штата не гарантировано. Повторю то, о чем уже говорил в главе 7: Copilot использует для своей работы только те данные, на которых обучался. Он не проверяет, правильны ли и актуальны ли возвращаемые им данные. Всегда выполняйте перекрестную проверку результатов, если вам нужны точные актуальные данные. Не следует предполагать, что значения, полученные от Copilot, актуальны и правильны.



Here is a mapping of all 50 U.S. states to area codes, with each state having a maximum of 10 area codes:

```
const stateAreaCodes = {
    AL: [205, 251, 256, 334, 938],
    AK: [907],
    AZ: [480, 520, 602, 623, 928],
    AR: [479, 501, 870],
    CA: [209, 213, 310, 323, 408, 415, 510, 530, 559, 562],
    CO: [303, 719, 720, 970],
    CT: [203, 475, 860, 959],
    DE: [302],
```

Рис. 8.6. Сгенерированные данные

Когда вы просите Copilot генерировать данные таким образом, вы можете иногда столкнуться с двумя другими типами проблем.

Во-первых, в зависимости от промпта и объема запрашиваемых данных Copilot может вернуть только часть данных. Обычно этого достаточно для начала, но набор не будет полным. Например, если исключить из запроса часть «все 50» и оставить только «Сопоставь штаты с телефонными кодами региона; ключом сделай сокращенное обозначение штата, а значением — массив кодов, содержащий не более 10 элементов», то Copilot может вернуть небольшую подборку записей:

```
state_area_codes = {
    "CA": [213, 310, 415, 510, 530, 559, 619, 626, 650, 661],
    "NY": [212, 315, 347, 516, 518, 585, 607, 631, 646, 716],
    "TX": [210, 214, 254, 281, 325, 346, 361, 409, 430, 432],
    "FL": [305, 321, 352, 386, 407, 561, 727, 754, 772, 786],
    "IL": [217, 224, 309, 312, 331, 618, 630, 708, 773, 815],
    "PA": [215, 267, 412, 484, 570, 610, 717, 724, 814, 878],
    "OH": [216, 234, 330, 380, 419, 440, 513, 567, 614, 740],
    "GA": [229, 404, 470, 478, 678, 706, 762, 770, 912, 943],
    "NC": [252, 336, 704, 743, 828, 910, 919, 980, 984, 336],
    "MI": [231, 248, 269, 313, 517, 586, 616, 734, 810, 906]
}
```

Если вы столкнетесь с подобной ситуацией, то можете попросить Copilot сгенерировать остальные данные, введя промпт «Сопоставь остальные штаты с телефонными кодами региона, ключом сделай сокращенное обозначение штата, а значением — массив кодов, содержащий не более 10 элементов». Мы добавили уточнение «остальные», чтобы Copilot продолжил с той точки, на которой остановился.

Во-вторых, при генерировании данных Copilot вы можете получить сообщение о том, что сгенерированные данные соответствуют данным в открытом репозитории. Если у вас включена настройка блокировки сгенерированного кода, который совпадает с кодом, находящимся в открытом доступе, то весь ответ Copilot на ваш запрос будет заблокирован.

Вероятность того, что такие данные являются чьей-то собственностью, невелика. Если о нарушении лицензии речь не идет, то промпт можно изменить, чтобы ИИ генерировал данные, используя другой параметр. Например, можно просто изменить объем запрашиваемых данных: «Сопоставь остальные штаты с телефонными кодами региона, ключом сделай сокращенное обозначение штата, а значением — массив кодов, содержащий не более 5 элементов». Обратите внимание: максимальное значение уменьшилось с 10 до 5.

Наконец, рассмотрим еще один способ взаимодействия с Copilot. Командная строка (CLI) GitHub встречается не так часто, как другие интерфейсы, о которых мы говорили в предыдущих главах, но тоже может использовать Copilot.

GitHub CLI и Copilot

Если вы работаете с GitHub CLI, то можете использовать подкоманды для взаимодействия с Copilot. На момент написания книги их было четыре.

Прежде чем использовать Copilot в GitHub CLI, необходимо выполнить несколько предварительных действий:

- установить GitHub CLI;
- выполнить аутентификацию на GitHub (например, с помощью команды `gh auth`);
- установить расширение Copilot в GitHub CLI (например, используя команду `gh extension install github/gh-copilot`).

Выполнив все эти действия, вы получите доступ к командам CLI. В табл. 8.1 указано назначение каждой из них.

Таблица 8.1. Команды Copilot в GitHub CLI

Команда	Назначение
`alias`	Генерирование псевдонимов/сокращений для конкретной оболочки (облегчает работу)
`config`	Настройка параметров
`explain`	Объяснение команды
`suggest`	Предложение команды

В качестве примера использования CLI с Copilot можно попросить его объяснить команду Unix с помощью простого вызова GitHub CLI, например, `gh copilot explain "ps -aux"`. Результат выглядит так:

```
Вас приветствует GitHub Copilot в CLI!
версия 1.1.0 (2025-02-10)

Я работаю на основе ИИ, поэтому возможны ошибки. Внимательно
проверьте весь сгенерированный код и предложения и поделитесь
обратной связью, чтобы я мог учиться и совершенствоваться.
Чтобы получить дополнительную информацию, обращайтесь по адресу:
 https://gh.io/gh-copilot-transparency

Объяснение:

  • ps используется для вывода информации о работающих процессах.
    • -aux является набором флагов:
      • -a выводит информацию обо всех процессах.
      • -u выводит подробную информацию о процессах.
      • -x включает процессы, не имеющие управляющего терминала.
```

В некоторых случаях Copilot предложит вам указать дополнительные уточняющие параметры. Например, если вы запросите у Copilot через CLI команду для установки Python, то ИИ может задать дополнительные вопросы, чтобы убедиться, что он предоставляет наилучший ответ. Вот вывод для исходной команды `gh copilot suggest "install python"`:

```
Вас приветствует GitHub Copilot в CLI!
версия 1.1.0 (2025-02-10)

...
? с какой командой я могу вам помочь?  [Используйте стрелки для перемещения,
 введите текст для фильтрации]
> общая команда оболочки
  gh команда
  git команда
```

```
Предложение:

  sudo apt-get install python3

? Выберите вариант [используйте стрелки для перемещения,
введите текст для фильтрации]
  Скопировать команду в буфер
> Объяснить команду
  Выполнить команду
  Переработать команду
  Оценить ответ
  Выйти

Объяснение:

  • sudo используется для выполнения команды с повышенными привилегиями,
    что делает возможным внесение изменений в системных файлах.
  • apt-get — команда управления пакетами для систем на базе Debian.
    • install — подкоманда для установки пакета.
      • python3 — конкретный устанавливаемый пакет, предоставляющий
                  интерпретатор Python 3.
```

Обратите внимание: выполнив исходную команду, Copilot вступил в дополнительное взаимодействие с вами. Он предложил вам выбрать контекст команды (общая оболочка, CLI или Git). Когда вы это сделали, Copilot выдал предложение. Затем он дал вам возможность скопировать команду, попросить объяснить ее, выполнить и т. д. После того как вы выбрали вариант «Объяснить», Copilot вывел такое же объяснение, как в предыдущем примере.

Резюме

В этой главе мы рассмотрели области, в которых поддержка со стороны ИИ является не самым очевидным вариантом, но может быть полезной. Возможности Copilot могут эффективно использоваться во всех этих областях.

Когда вы взаимодействуете с базами данных, Copilot может предоставить предложения по завершению кода и генерировать готовые конструкции для SQL, а также для более традиционных языков программирования. Copilot может быть особенно полезен для создания хранимых процедур и оптимизации кода.

Помимо генерирования традиционного кода, который сообщает системе, как что-то сделать, Copilot может создавать декларативные спецификации, описывающие желаемый конечный результат. Хороший пример — использование Copilot для объявления спецификаций Kubernetes. При наличии хорошо написанного промпта Copilot может сгенерировать определение YAML для любого стандартного объекта Kubernetes. Кроме того, Copilot может сообщить пользователю, как применять спецификацию с Kubernetes и даже преобразовать ее в код, используя вызовы API. Такая помощь сильно упрощает работу со сложными особенностями разных фреймворков и их представлениями в YAML.

Кроме того, Copilot позволяет автоматизировать другую задачу, которая часто вызывает затруднения у разработчиков, — создание регулярных выражений. При наличии хорошо написанного промпта Copilot сгенерирует необходимую комбинацию символов для любого поиска совпадений, который вам нужен, и приведет примеры использования этого выражения.

Для поддержки базового тестирования или демонстраций Copilot может создать набор данных, с которым будет работать код. Это могут быть сопоставления ключей и значений, определенные с помощью промпта или из контекста. Правильность или актуальность данных не гарантируется, поэтому при работе с результатами необходимо быть особенно внимательными.

Наконец, мы рассмотрели возможности, добавленные в GitHub CLI для работы с Copilot. Используя GitHub CLI, Copilot может предлагать команды для выполнения таких операций, как установка приложений, а также объяснять назначение команд и аргументов. Эта функциональность ограничена, но есть вероятность, что в будущем она будет расширяться.

В следующей главе вы узнаете, как возможности Copilot интегрируются в другое базовое приложение, которое используют разработчики, — GitHub.

ГЛАВА 9

Использование Copilot в GitHub

Как и многие другие ключевые средства разработки, ИИ был интегрирован в экосистему GitHub через Copilot. Это означает, что вы можете общаться с Copilot непосредственно в GitHub, а также использовать возможности ИИ, чтобы работать напрямую с такой ключевой функциональностью, как pull-запросы и запросы на решение проблем.

В этой главе вы увидите, как Copilot интегрируется в GitHub, и узнаете, как эффективно использовать его возможности при работе на этой платформе. Мы обсудим следующие темы:

- использование Copilot Chat с репозиториями GitHub;
- использование Copilot для изменения процессов разработки кода;
- Copilot и pull-запросы;
- изучение изменений в коде с помощью Copilot;
- использование Copilot с задачами (issues) GitHub.

Изменение интерфейса GitHub

Как и вся другая информация в этой книге, снимки экрана и сведения об элементах интерфейса и их расположении актуальны на момент ее написания. Как и интеграция Copilot в IDE, интеграция в GitHub может изменяться со временем.

Для начала посмотрим, как Copilot интегрируется с вашими репозиториями в GitHub.

Использование чата с репозиториями GitHub

Чат-интерфейс наподобие тех, которые доступны в IDE для интеграции с Copilot, доступен и в GitHub. Общий интерфейс доступен по адресу https://github.com/copilot. Кроме того, на странице Code ваших репозиториев в верхней строке элементов управления находится значок Copilot (рис. 9.1). При щелчке на нем открывается полноэкранный чат Copilot для данного проекта. Вскоре мы вернемся к нему.

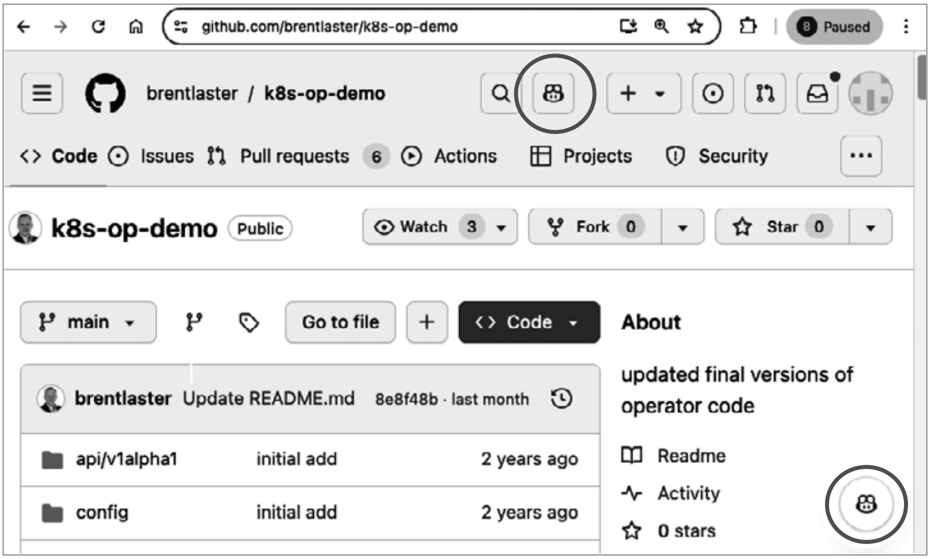


Рис. 9.1. Элементы Copilot на странице репозитория

Если щелкнуть на стрелке рядом со значком Copilot, то открывается меню (также показанное на рис. 9.1). В нем нас интересует команда Assistive.

При выборе этой команды открывается чат-интерфейс, заранее заполненный возможными вопросами о выбранном репозитории (рис. 9.2).

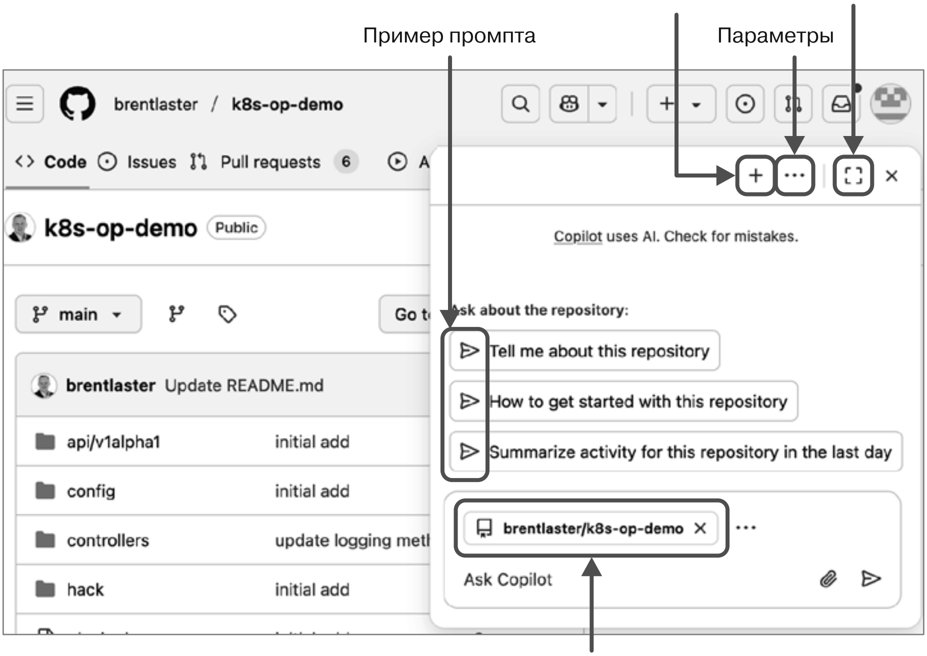
Контекст чата в GitHub

Новые экземпляры чата Assistive заполняются вопросами, относящимися к элементу GitHub, активному в момент открытия. Например, если вы работаете над проблемой или задачей (issue) GitHub и открываете новый чат, он содержит заранее подготовленные вопросы и указания, такие как «Выдай общее описание проблемы» и «Что делать дальше?». Если вы выберете один из них, то чат заполнится гораздо более длинным промптом, специально разработанным для получения более подробной информации.

Если выбрать первое указание, в котором мы просим Copilot рассказать о репозитории, то Copilot получает готовый промпт, приведенный ниже:

```
Дай ответы к следующим заданиям в отдельных абзацах:

 - Предоставь информацию о предназначении репозитория. Предоставь также
   сводку содержимого файла README, если он есть.
 - Предоставь подробный анализ возможностей и технологий,
   использованных в этом репозитории, если он содержит реализацию
   программной системы любого типа. В противном случае предоставь
   анализ содержимого этого репозитория.
```

```
Дополнительные указания:
 - Предоставь для каждого абзаца заголовок, выделенный жирным шрифтом.
 - Анализ должен быть кратким и написанным в разговорном стиле.
 - Не повторяй и не перечисляй подробности контекста в ответе.
 - Предоставь краткий анализ информации, избегая лишних повторений.
```



Продолжить чат в полноэкранном режиме

Новый чат

Параметры

Пример промпта

Репозиторий как контекст по умолчанию

Рис. 9.2. Вопросы о репозитории в чате Assistive

Ответ Copilot на промпт выглядит так:

```
**Предназначение репозитория**

Этот репозиторий предоставляет базовый пример реализации оператора
Kubernetes. Он служит обучающим ресурсом для создания операторов
Kubernetes с использованием стандартных инструментариев.
В файле README объясняется, как создать и запустить оператор, и описываются
действия по его развертыванию в кластере Kubernetes, тестированию
и изменению определений API. Проект разработан в соответствии с паттерном
Kubernetes "Оператор", цель которого — синхронизация ресурсов
с их желаемым состоянием.
```

```
Использованная функциональность и технологии

Репозиторий создается на основе Go и использует принципы
разработки операторов Kubernetes. Он использует Kubebuilder для создания
контроллеров, разработанных в соответствии с паттерном Kubernetes "Оператор".
Оператор взаимодействует с кластерами Kubernetes, управляя
CRD (Custom Resource Definitions) и обеспечивая соответствие
состояния кластера нужной конфигурации. Кроме того, репозиторий
предоставляет Makefile-цели для таких задач, как создание,
развертывание и удаление оператора.
```

Такая информация может быть особенно полезной для новых участников команды или для специалистов, которые должны принять ответственность за кодовую базу.

Если вы предпочитаете полноэкранный режим работы с чатом, то нажмите квадратную кнопку в правом верхнем углу, чтобы перейти к полноэкранному сеансу чата. Полноэкранный режим напоминает чат-интерфейс IDE, в котором репозиторий задан в качестве контекста по умолчанию. Этот режим открывается в том же экране, который вы открываете при щелчке на значке Copilot (а не на стрелке рядом), как было показано на рис. 9.1.

При щелчке на значке или переходе от экрана Assistive к полноэкранному режиму в обоих случаях открывается страница https://github.com/copilot. Чат-интерфейс этой страницы назначает в качестве контекста по умолчанию текущий репозиторий. На рис. 9.3 показан полноэкранный режим для Copilot Chat в GitHub.

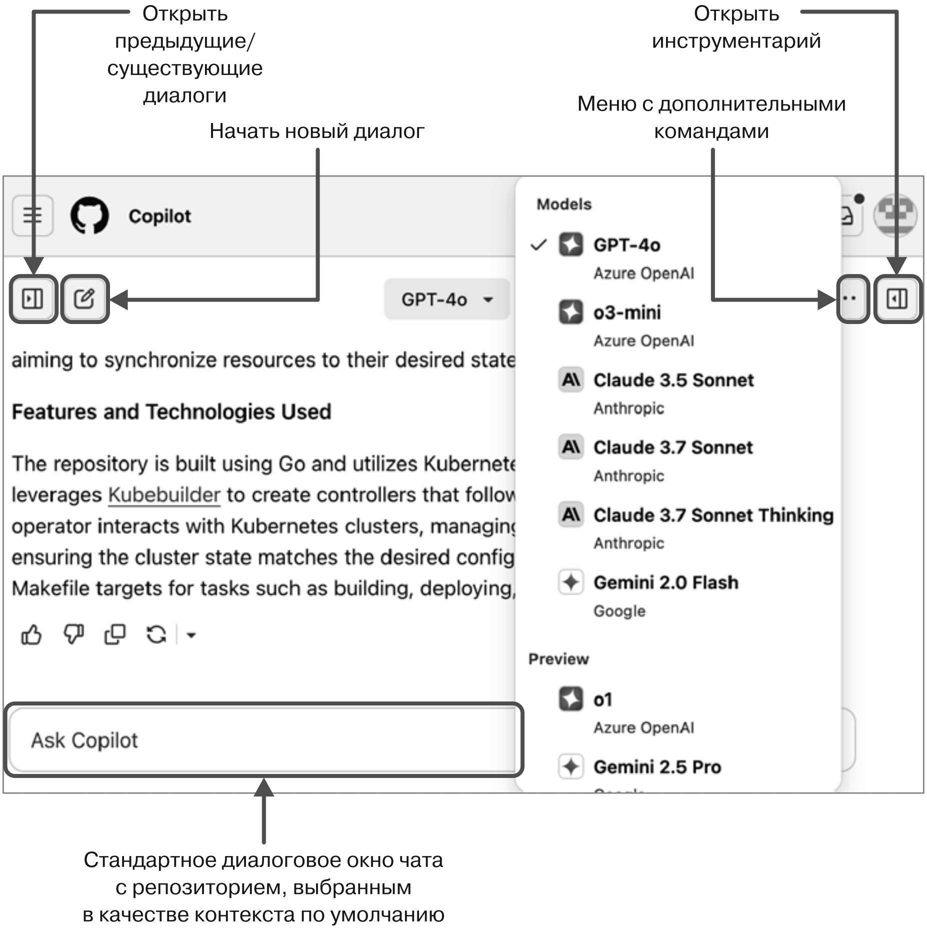
Элементы главной области ввода текста диалогового окна работают практически так же, как элементы соответствующей области диалогового окна чата в IDE. Одно из различий заключается в том, что при нажатии кнопки со стрелкой Submit не появляются дополнительные варианты, так как у вас нет участников чата или других элементов, которым может отправляться запрос.

Теперь обсудим другие элементы и варианты, доступные в диалоговом окне чата в GitHub. (Некоторые из них могут быть невидимыми, пока вы не перейдете в полноэкранный режим.)

При выборе элемента в левом верхнем углу (квадрат с полоской и стрелкой, направленной вправо) открывается список всех диалогов, которые вы проводили в чате, и вы можете выбирать их и переключаться между ними.

При выборе следующего элемента справа (квадрата с карандашом) открывается новый диалог.

Крайний правый элемент обозначен как некий инструментарий (Open Workbench). На момент написания книги не было никакой реальной документации относительно того, что должна делать эта функция.



aiming to synchronize resources to their desired state

Features and Technologies Used

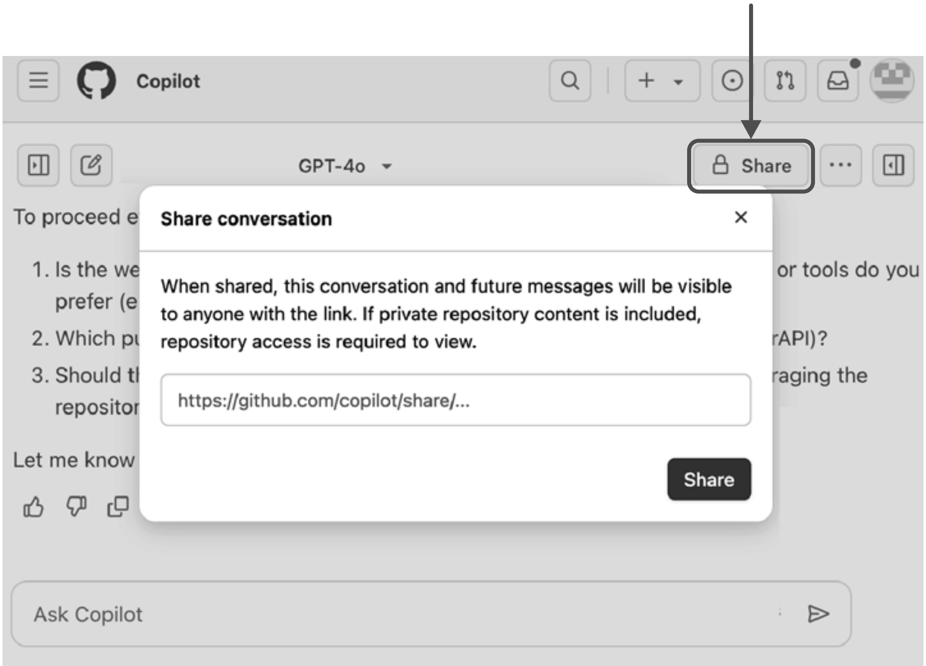
The repository is built using Go and utilizes Kubernete
leverages Kubebuilder to create controllers that follow
operator interacts with Kubernetes clusters, managin
ensuring the cluster state matches the desired config
Makefile targets for tasks such as building, deploying,

Рис. 9.3. Полноэкранный режим чата

Когда вы введете первый промпт, в чате появится еще один элемент: кнопка Share. Она упрощает совместное использование чата Copilot для таких целей, как программирование с напарником/диагностика, добавление информации для pull-запросов и т. д.

Чтобы предоставить доступ к текущему диалогу и будущим сообщениям, нажмите кнопку Share в правом верхнем углу, а затем снова нажмите Share в открывшемся диалоговом окне (рис. 9.4).

После этого можно щелкнуть на стрелке, направленной вниз, рядом с Share, и скопировать ссылку на диалог, которую вы можете передать другим разработчикам (рис. 9.5). При желании можно закрыть доступ к диалогу.


Кнопка Share появляется после введения промпта и открытия диалога

To proceed e

1. Is the we … prefer (e … or tools do you
2. Which pu … rAPI)?
3. Should tl … repositor … raging the

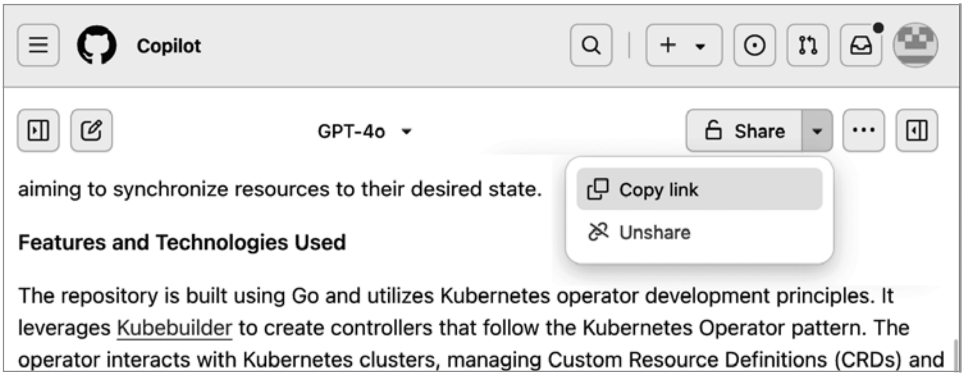
Let me know

Share conversation

When shared, this conversation and future messages will be visible to anyone with the link. If private repository content is included, repository access is required to view.

https://github.com/copilot/share/...

Рис. 9.4. Операция предоставления доступа



aiming to synchronize resources to their desired state.

Features and Technologies Used

The repository is built using Go and utilizes Kubernetes operator development principles. It leverages Kubebuilder to create controllers that follow the Kubernetes Operator pattern. The operator interacts with Kubernetes clusters, managing Custom Resource Definitions (CRDs) and

Рис. 9.5. Получение ссылки на диалог

Наконец, при щелчке на элементе вызова меню с многоточием *(справа)* открывается меню для доступа к расширенным функциям (рис. 9.6).

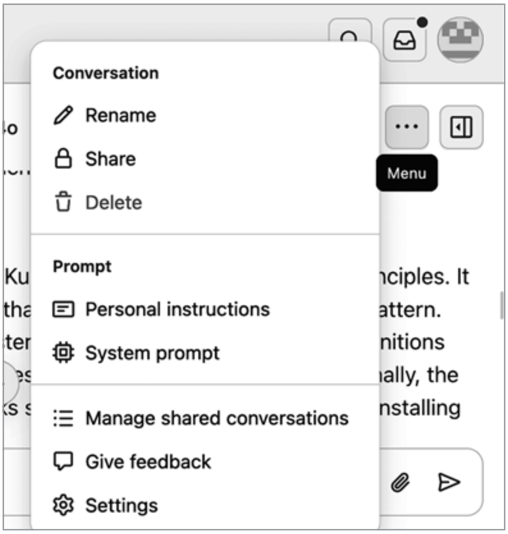


Рис. 9.6. Меню чата

В меню доступны следующие команды.

Conversation

Команды в этом разделе не требуют пояснений: они предназначены для переименования, удаления или предоставления доступа к текущему диалогу.

Personal instructions

Команды позволяют настроить реакцию Copilot Chat на ваши запросы на GitHub. При выборе команды открывается диалоговое окно, в котором можно задать такие предпочтения, как язык, тон общения или стиль программирования (например, «Всегда давай ответы на Python» или «Высказывайся кратко»). Эти инструкции имеют приоритет над всеми настройками уровня репозитория или организации и применяются ко всем сеансам Copilot Chat на GitHub.

System prompt

Эта команда выводит текст системного промпта, заданного для Copilot на GitHub.

Manage shared conversations

Открывает список диалогов, к которым вы предоставили доступ, и позволяет получить ссылку или закрыть доступ к каждому диалогу.

Give feedback

Объяснений не требует.

Settings

Выбрав эту команду, вы перейдете к вашим персональным настройкам для Copilot.

Чат-интерфейс служит не только для общения с ИИ по поводу репозитория, но и запускает процесс его индексирования. Ранее мы обсуждали, как индексы создаются и используются в IDE с помощью Copilot. Для репозиториев на стороне GitHub создаются отдельные индексы. Но цель остается прежней: помочь Copilot получить гораздо более глубокое и расширенное представление о репозитории. О том, как работает индексирование репозиториев GitHub, более подробно рассказано во врезке.

ИНДЕКСИРОВАНИЕ В GITHUB

Когда вы открываете GitHub Copilot Chat в репозитории на GitHub, этот репозиторий автоматически индексируется в фоновом режиме. Этот процесс позволяет Copilot Chat предоставлять ответы о вашей кодовой базе, зависящие от контекста, — например, объяснять структуру кода или логику, ссылаясь на соответствующие разделы вашего репозитория. Запуск индексирования не требует выполнения никаких действий вручную — просто откройте Copilot Chat, выбрав репозиторий в качестве контекста.

Вот ключевые характеристики, связанные с индексированием.

Автоматический запуск

Индексирование автоматически запускается, когда вы открываете Copilot Chat на GitHub, выбрав репозиторий в качестве контекста.

Скорость

Исходное индексирование выполняется быстро, от нескольких секунд до минуты — даже для больших репозиториев.

Непрерывные обновления

После индексирования индекс репозитория автоматически обновляется, как правило, в течение нескольких секунд после запуска нового диалога.

Отсутствие ограничений по индексированию

Количество индексируемых репозиториев не ограничено, причем это относится ко всем тарифным планам Copilot, включая бесплатный.

Отсутствие обучения модели

Индексированные данные не используются для обучения модели; они нужны только для улучшения способности Copilot отвечать на вопросы о коде в вашем репозитории.

Управление доступом

Только пользователи с доступом к репозиторию могут использовать его индексированный контекст в Copilot Chat.

Чат-интерфейс и функциональность индексирования помогают понять и более глубоко изучить репозитории и другие объекты в GitHub. В частности, это может быть полезно для специалистов, которые присоединяются к новому проекту или принимают ответственность за внесение изменений и исправлений.

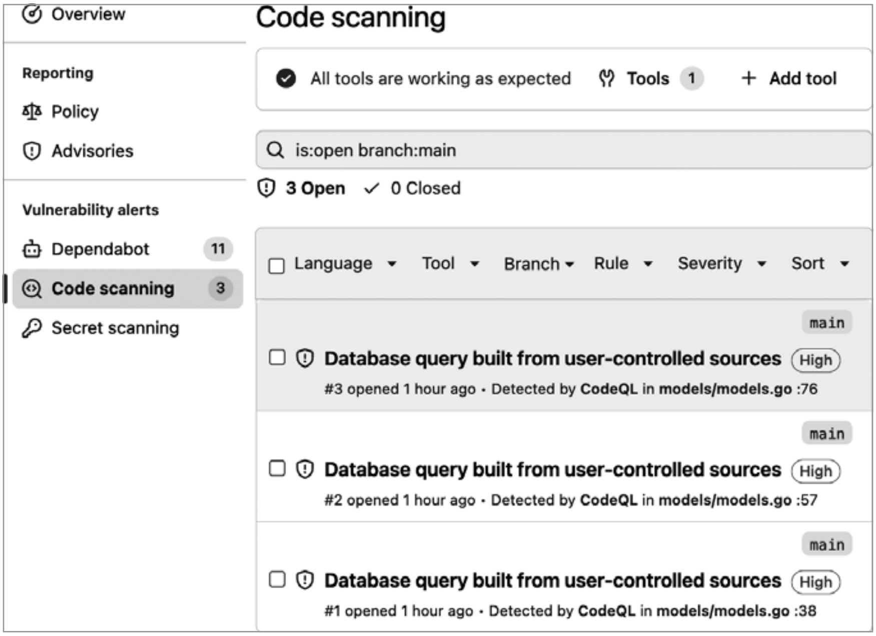
Но в области работы с изменениями и исправлениями в GitHub можно использовать Copilot другими способами, чтобы сделать процессы разработки кода более эффективными и менее трудоемкими.

Использование Copilot для изменения процессов разработки кода

Copilot предоставляет полезные возможности для изменения кода, начиная с предложений по исправлению, помощи в исправлении проблем при обновлении и pull-запросов и заканчивая ревью кода. В этом разделе мы проанализируем несколько проблем и способы их решения с помощью Copilot.

Для начала рассмотрим пример с проблемой безопасности. У нас есть код с набором запросов к базе данных, которые из-за особенностей своей структуры уязвимы для внедрения SQL. Такие проблемы легко выявить, просканировав код на GitHub с помощью CodeQL (https://codeql.github.com).

Оповещения о проблемах при сканировании кода в проблемном репозитории показаны на рис. 9.7.



All tools are working as expected

is:open branch:main

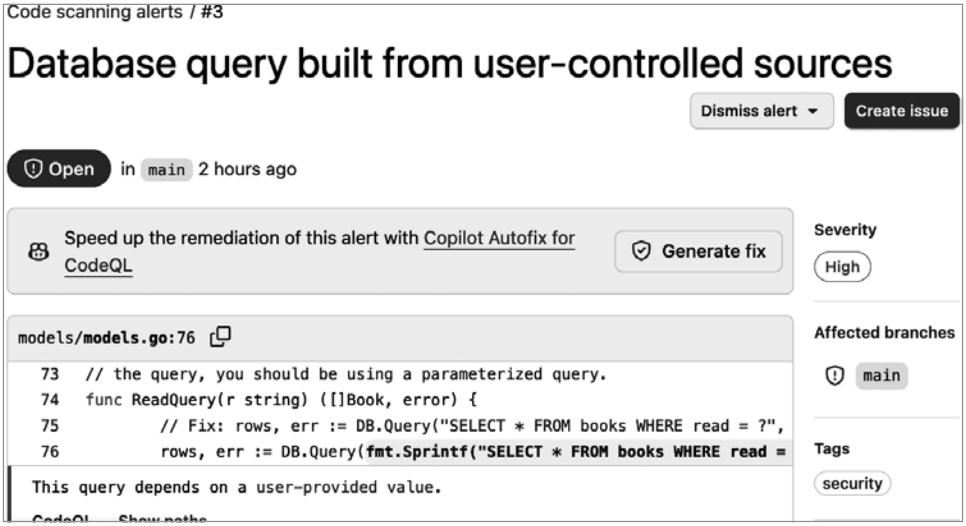
☐ Database query built from user-controlled sources High
#3 opened 1 hour ago · Detected by CodeQL in models/models.go :76

☐ Database query built from user-controlled sources High
#2 opened 1 hour ago · Detected by CodeQL in models/models.go :57

☐ Database query built from user-controlled sources High
#1 opened 1 hour ago · Detected by CodeQL in models/models.go :38

Рис. 9.7. Проблемы при сканировании кода

Выберем одну из этих проблем и поработаем над ней. Пусть это будет последняя проблема. Открывая ссылку на панели Code scanning, вы получите более подробную информацию о конкретном оповещении (рис. 9.8).



Database query built from user-controlled sources

Open in `main` 2 hours ago

Speed up the remediation of this alert with Copilot Autofix for CodeQL

Severity: High

Affected branches: main

Tags: security

models/**models.go**:76

```
// the query, you should be using a parameterized query.
func ReadQuery(r string) ([]Book, error) {
        // Fix: rows, err := DB.Query("SELECT * FROM books WHERE read = ?",
        rows, err := DB.Query(fmt.Sprintf("SELECT * FROM books WHERE read =
```

This query depends on a user-provided value.

Рис. 9.8. Оповещение при сканировании кода

Было бы неплохо, если бы Copilot мог выдать сводку по оповещению. К сожалению, на момент написания книги он не имел такой возможности. Однако на том же экране есть кнопка Create issue для создания сообщения о проблеме (справа наверху на рис. 9.8).

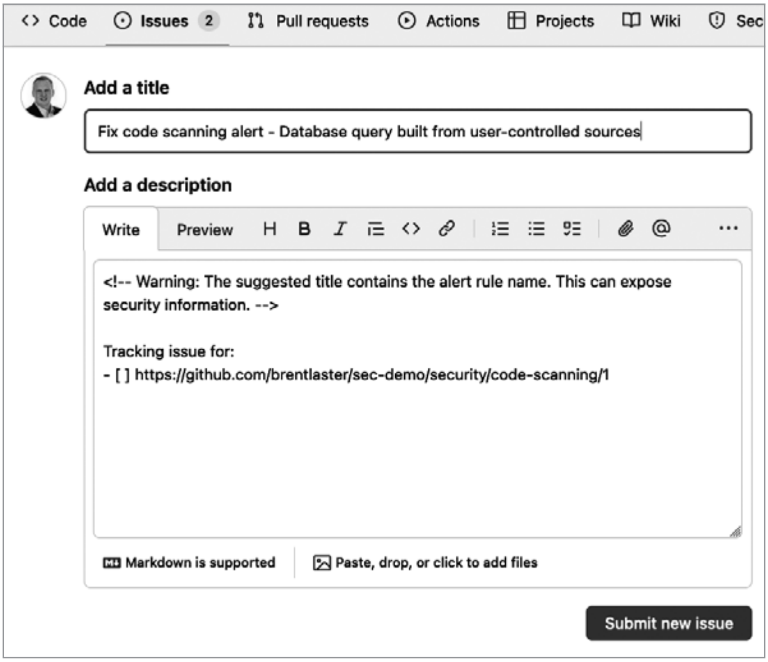
Включение сообщений о проблемах

Кнопка Create issue доступна только в том случае, если сообщения о проблемах включены в настройках репозитория.

Нажав кнопку Create issue, вы создадите базовую задачу GitHub Issue с заголовком и ссылкой на оповещение, выданное при сканировании (рис. 9.9). (При этом появляется предупреждение, которое в целом полезно, но для этого примера некритично.)

Нажав кнопку Submit new issue, вы завершите создание задачи (рис. 9.10).

Если после этого вы вернетесь к странице оповещения (рис. 9.11), то увидите, что на ней появилась пометка Tracked by #3, в которой #3 означает задачу GitHub Issue № 3 в этом репозитории.



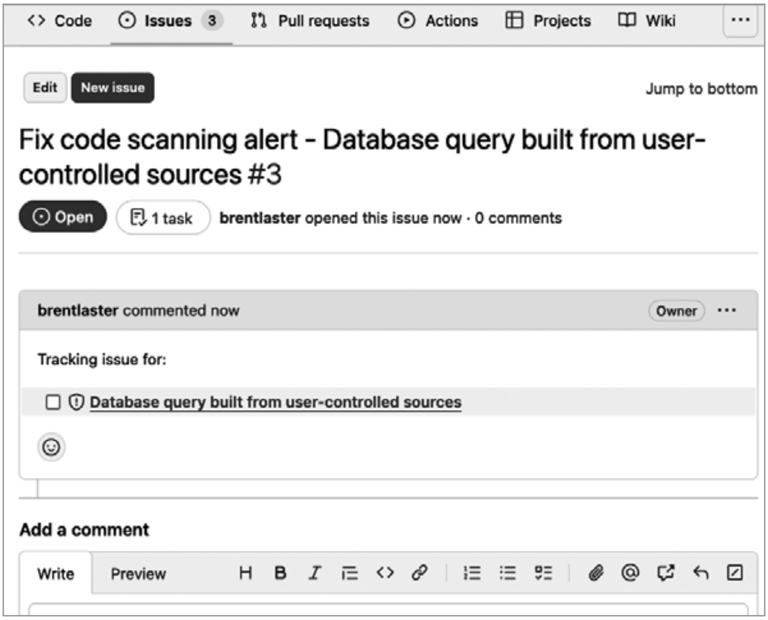
Add a title

Fix code scanning alert - Database query built from user-controlled sources

Add a description

```
<!-- Warning: The suggested title contains the alert rule name. This can expose
security information. -->

Tracking issue for:
- [ ] https://github.com/brentlaster/sec-demo/security/code-scanning/1
```

Рис. 9.9. Сгенерированный черновик задачи



Fix code scanning alert - Database query built from user-controlled sources #3

brentlaster opened this issue now · 0 comments

brentlaster commented now

Tracking issue for:

☐ Database query built from user-controlled sources

Add a comment

Рис. 9.10. Сгенерированная задача

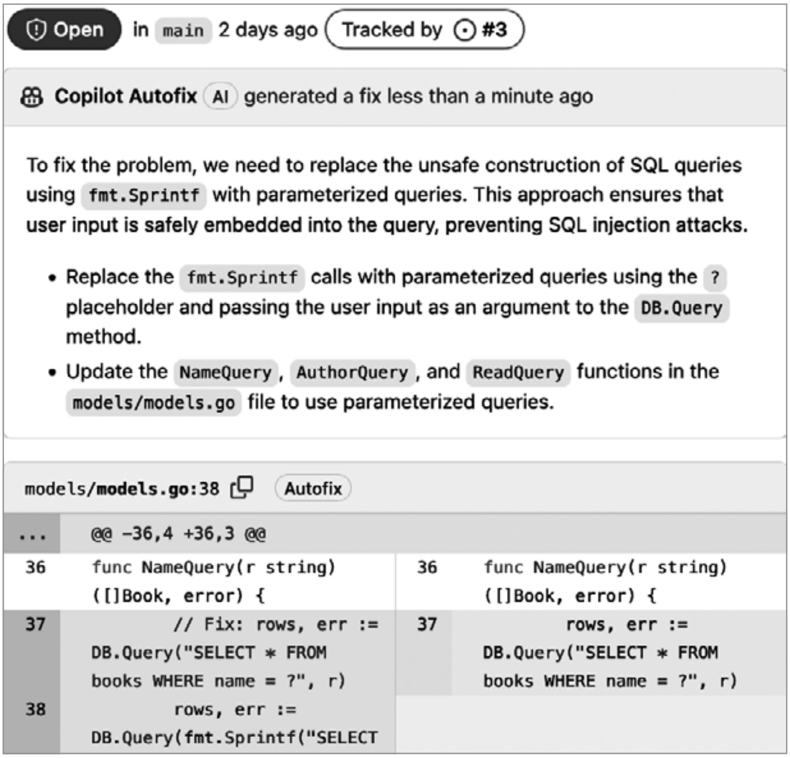
Database query built from user-controlled sources

Speed up the remediation of this alert with Copilot Autofix for CodeQL

Рис. 9.11. Страница оповещения

Возможно, вы заметили на рисунке баннер с надписью **Copilot Autofix for CodeQL** и кнопкой **Generate fix**. *Copilot Autofix* — это функция, которая интегрируется со средствами сканирования кода GitHub (такими как CodeQL) для анализа кода, выявления проблем и создания предложений по их исправлению с использованием Copilot.

Если вы нажмете кнопку **Generate fix**, то Copilot предложит внести в код изменения, чтобы исправить проблему (рис. 9.12).



To fix the problem, we need to replace the unsafe construction of SQL queries using `fmt.Sprintf` with parameterized queries. This approach ensures that user input is safely embedded into the query, preventing SQL injection attacks.

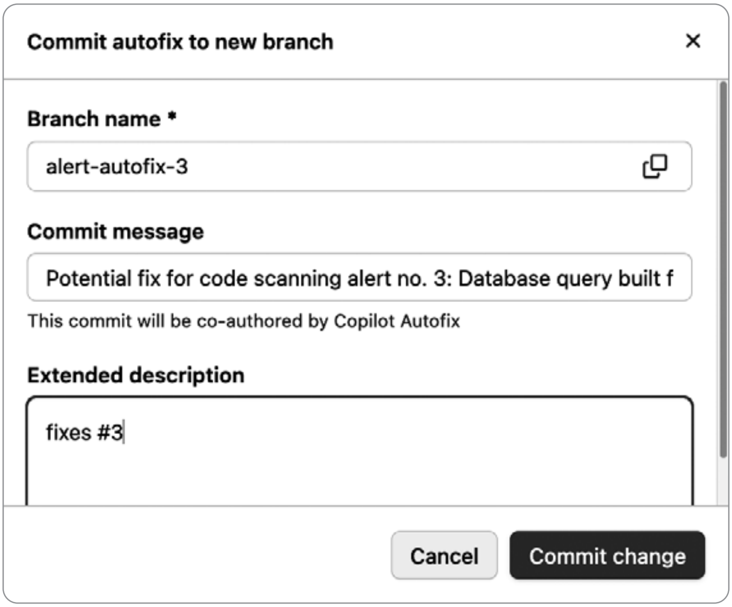
- Replace the `fmt.Sprintf` calls with parameterized queries using the `?` placeholder and passing the user input as an argument to the `DB.Query` method.
- Update the `NameQuery`, `AuthorQuery`, and `ReadQuery` functions in the `models/models.go` file to use parameterized queries.

```
@@ -36,4 +36,3 @@
func NameQuery(r string) ([]Book, error) {
        // Fix: rows, err := DB.Query("SELECT * FROM books WHERE name = ?", r)
        rows, err := DB.Query(fmt.Sprintf("SELECT
```

```
func NameQuery(r string) ([]Book, error) {
        rows, err := DB.Query("SELECT * FROM books WHERE name = ?", r)
```

Рис. 9.12. Предложенное исправление от Copilot

В конце списка предлагаемых изменений кода находится кнопка Commit change. Если ее нажать, то изменения сохраняются в новой ветке (по умолчанию), при этом автоматически генерируются значения для имени ветки и сообщения о коммите. Чтобы убедиться в том, что оно связано с проблемой, открытой для этого изменения, мы добавили расширенное описание `fixes #3`. Диалоговое окно коммита показано на рис. 9.13.



Branch name *

alert-autofix-3

Commit message

Potential fix for code scanning alert no. 3: Database query built f

This commit will be co-authored by Copilot Autofix

Extended description

fixes #3

Рис. 9.13. Подготовка коммита для автоматического исправления

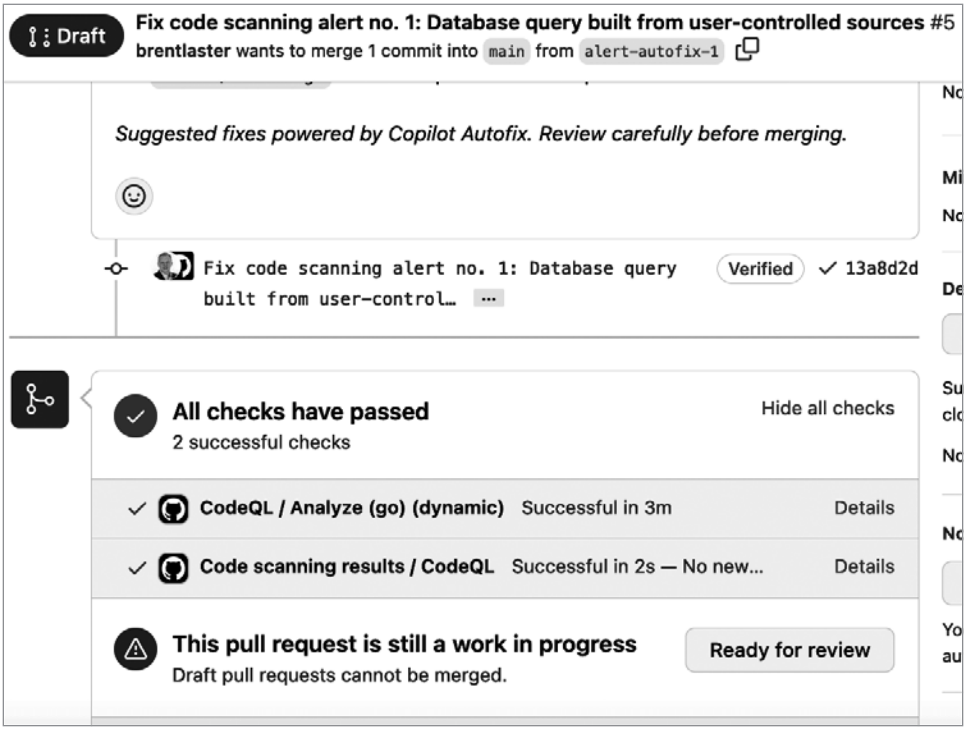
Автоматическое завершение коммита открывает черновик pull-запроса, в рамках которого CodeQL запускается повторно, чтобы проверить, исправлены ли проблемы. В следующем разделе мы на примере этого изменения более подробно разберем, как с помощью Copilot управлять pull-запросами.

Использование Copilot для управления pull-запросами

Pull-запросы GitHub — проверенный временем метод управления слияниями в GitHib между разными репозиториями или между ветками одного репозитория. Pull-запросы предоставляют возможность совместной работы по объединению или проверке изменений из исходной ветки. Кроме того, они защищают цель слияния от плохого кода, используя такие средства, как ревью кода и запуски рабочих процессов GitHub Actions.

С pull-запросами связаны и некоторые сложности. Одна из них — получение информации об изменениях и диалогах, происходящих в pull-запросе в процессе работы над ним. К счастью, Copilot может с этим помочь.

Вернемся к примеру с проблемой безопасности в нашем репозитории, которые были устранены в другой ветке. Теперь мы хотим перенести эти исправления в рабочую ветку (`main` в нашем случае), используя процесс pull-запросов.



Fix code scanning alert no. 1: Database query built from user-controlled sources #5

brentlaster wants to merge 1 commit into `main` from `alert-autofix-1`

Suggested fixes powered by Copilot Autofix. Review carefully before merging.

`Fix code scanning alert no. 1: Database query built from user-control…` ✓ 13a8d2d

All checks have passed

2 successful checks

✓ **CodeQL / Analyze (go) (dynamic)** Successful in 3m

✓ **Code scanning results / CodeQL** Successful in 2s — No new...

This pull request is still a work in progress

Draft pull requests cannot be merged.

Рис. 9.14. Черновик pull-запроса, созданный функцией автоматического исправления

Когда подготовка завершится, черновой pull-запрос можно преобразовать в запрос, готовый к рецензированию; для этого нужно нажать кнопку Ready for review. После этого можно пройти стандартные этапы работы с pull-запросом, включая добавление рецензентов.

Рецензирование pull-запросов с помощью Copilot

Copilot можно добавить в качестве автоматизированного инструмента рецензирования для получения обратной связи о внесенных изменениях. Это делается так: в разделе Reviewers pull-запроса щелкните на поле для выбора пользователя и увидите Copilot в списке доступных рецензентов (рис. 9.15).

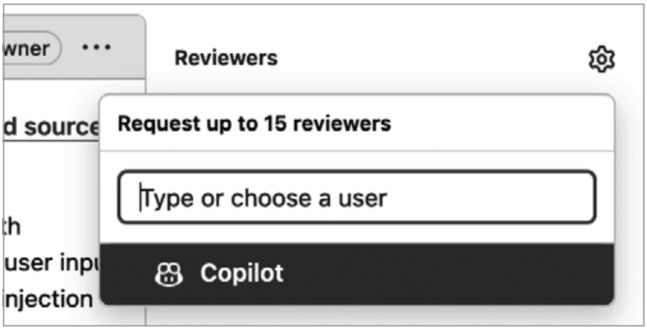
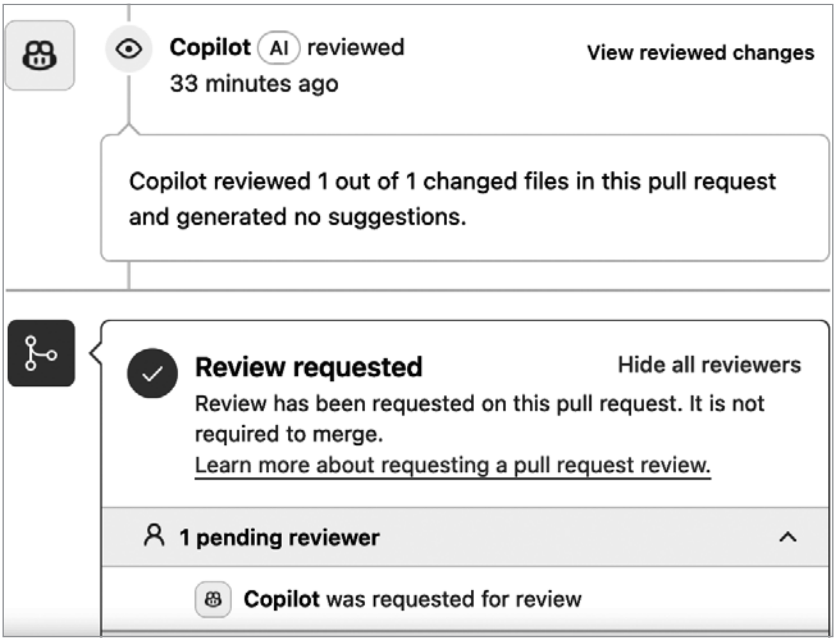


Рис. 9.15. Добавление Copilot в качестве рецензента для pull-запроса

Выполнив ревью кода, Copilot опубликует результаты в pull-запросе (рис. 9.16). В данном случае Copilot не сделал никаких замечаний. И это вполне логично, поскольку внести исправление предложил именно он.



Copilot reviewed 1 out of 1 changed files in this pull request and generated no suggestions.

Review has been requested on this pull request. It is not required to merge.
Learn more about requesting a pull request review.

Copilot was requested for review

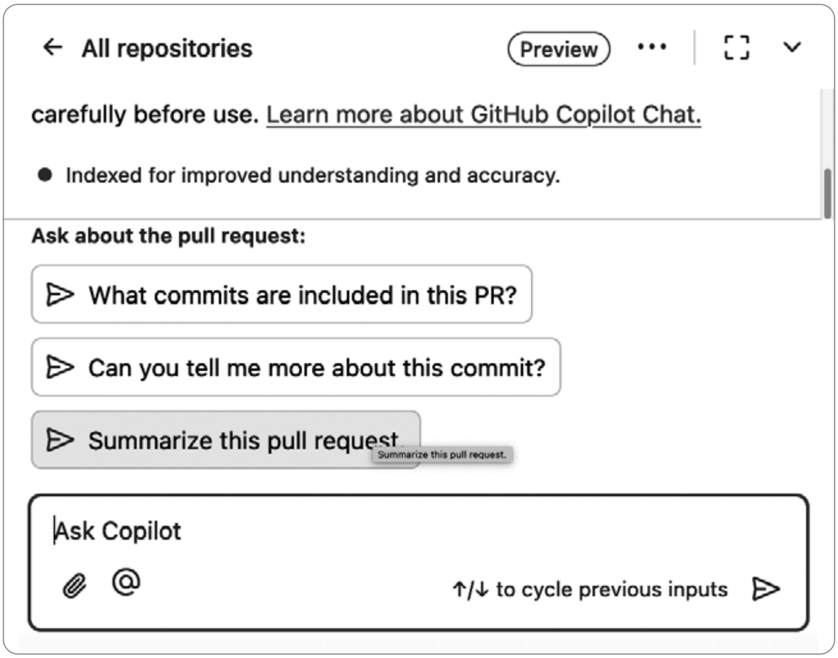
Рис. 9.16. Результат рецензирования, выполненного Copilot

На данном этапе все готово к слиянию pull-запроса. В качестве завершающей проверки (или если вам нужно предоставить объяснение другим разработчикам) вы можете попросить Copilot выдать сводное описание этого pull-запроса.

Используйте кнопки для открытия чата Copilot: одна находится в верхней части страницы, а другая — *плавающая* — обычно внизу справа (рис. 9.17).

Рис. 9.17. Разные способы открытия чата Copilot

Используйте любой из этих способов, чтобы открыть сеанс чата. Затем можно попросить Copilot выполнить задачи, связанные с pull-запросом, например: «Предоставь сводное описание этого pull-запроса» (рис. 9.18).

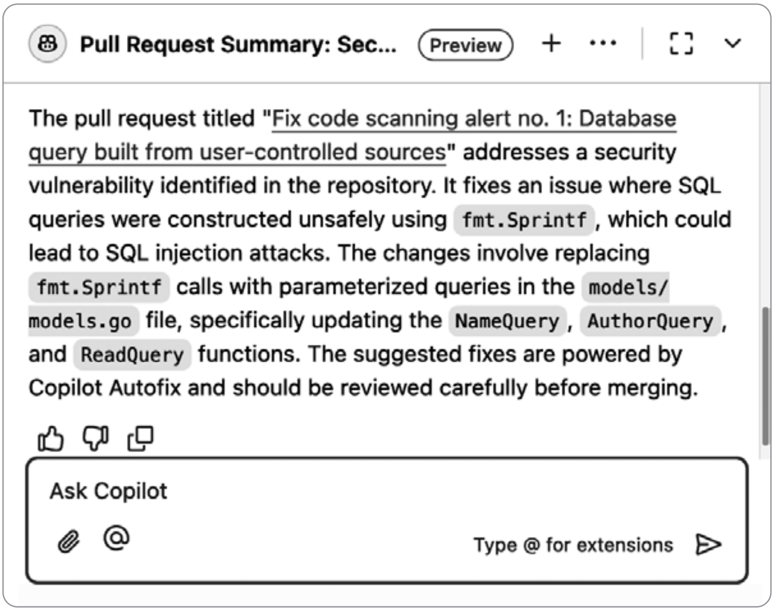

carefully before use. Learn more about GitHub Copilot Chat.

● Indexed for improved understanding and accuracy.

Ask about the pull request:

Рис. 9.18. Copilot получает указание выдать сводное описание pull-запроса

Copilot выдает короткую и понятную сводку содержания pull-запроса (рис. 9.19).



The pull request titled "Fix code scanning alert no. 1: Database query built from user-controlled sources" addresses a security vulnerability identified in the repository. It fixes an issue where SQL queries were constructed unsafely using `fmt.Sprintf`, which could lead to SQL injection attacks. The changes involve replacing `fmt.Sprintf` calls with parameterized queries in the `models/models.go` file, specifically updating the `NameQuery`, `AuthorQuery`, and `ReadQuery` functions. The suggested fixes are powered by Copilot Autofix and should be reviewed carefully before merging.

Рис. 9.19. Сводное описание pull-запроса

Рассмотрим еще один пример того, как Copilot может помочь с pull-запросами, — ускоренный способ их создания.

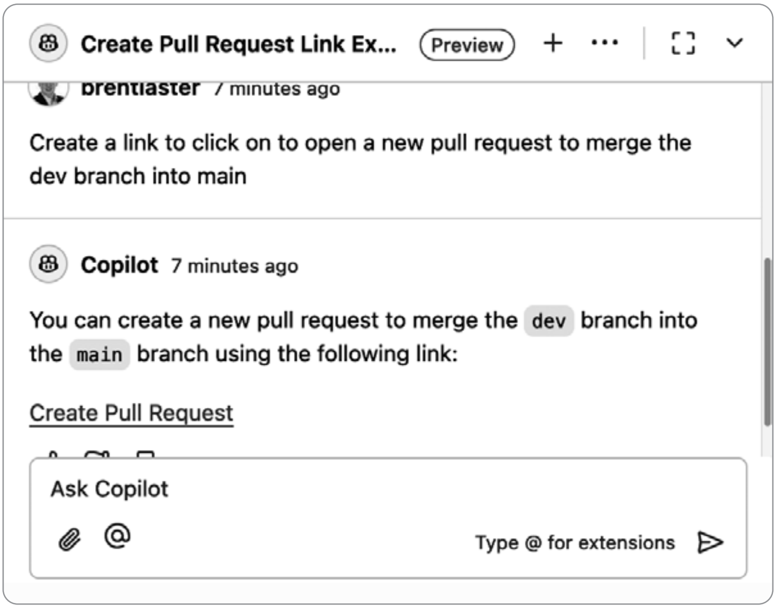
Использование Copilot для упрощения запуска pull-запросов

Допустим, в том же репозитории, в котором вы работали, вы создали исправления для *всех трех* оповещений и отправили их в другую ветку — `dev`. Представим также, что вы хотите создать новый pull-запрос для слияния веток `dev` и `main`.

Конечно, можно пройти по меню pull-запросов и выполнить действия по созданию нового запроса. Но благодаря Copilot этот процесс можно упростить до одного щелчка. Большинство действий в GitHub можно выполнить, перейдя на определенный URL, поэтому можно попросить Copilot сгенерировать подходящий адрес. Для этого нужно открыть чат-интерфейс Copilot и ввести такой промпт:

```
Создай ссылку, по щелчку на которой открывается новый pull-запрос
для слияния веток dev и main.
```

Copilot выводит в чате ссылку (рис. 9.20). Обратите внимание: он всего лишь генерирует ссылку, которую вы могли бы ввести и сами.



brentlaster 7 minutes ago

Create a link to click on to open a new pull request to merge the dev branch into main

Copilot 7 minutes ago

You can create a new pull request to merge the `dev` branch into the `main` branch using the following link:

Create Pull Request

Рис. 9.20. Сгенерированная ссылка для pull-запроса

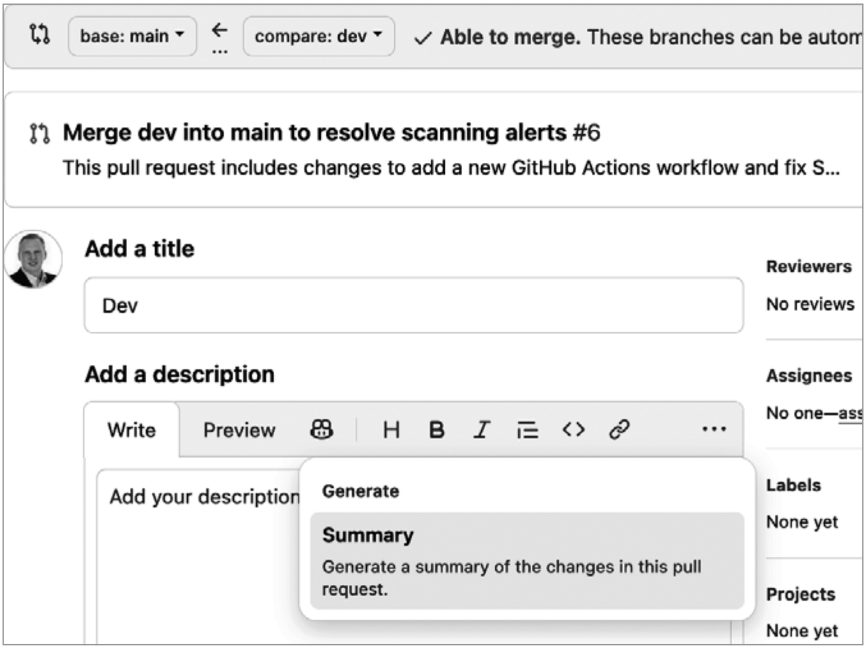
Если щелкнуть на сгенерированной ссылке, то откроется новый pull-запрос с уже настроенными компонентами `base` и `compare`. Таким образом, Copilot помогает упростить создание pull-запросов. Но, помимо этого, он может сгенерировать краткое описание такого запроса, основываясь на внесенных изменениях. На рис. 9.21 показано, как это сделать: щелкните на значке Copilot в области описания, а затем выберите команду для создания описания. (Обратите внимание: на момент написания книги эта команда доступна только в платных тарифных планах Copilot.)

Когда вы щелкнете на команде, Copilot сгенерирует описание изменений в соответствующем поле. Информация выводится в формате Markdown, и ее можно редактировать. На рис. 9.22 показан пример того, как выглядит это описание после создания pull-запроса. Комментарии будут отображаться как ваши, хотя на самом деле их сгенерировал ИИ. Кроме того, описание будет содержать подробную информацию о каждом изменении (а также ссылки на измененные файлы).

Генерирование URL для других задач

В этом разделе мы попросили Copilot создать ссылку для нового pull-запроса, но аналогичный подход можно использовать для любых действий, для которых GitHub предоставляет соответствующий прямой URL.

Помимо всего вышесказанного, Copilot можно использовать для более глубокого изучения подробностей изменений в коде.



✓ **Able to merge.** These branches can be autom

Merge dev into main to resolve scanning alerts #6

This pull request includes changes to add a new GitHub Actions workflow and fix S...

Add a title

Dev

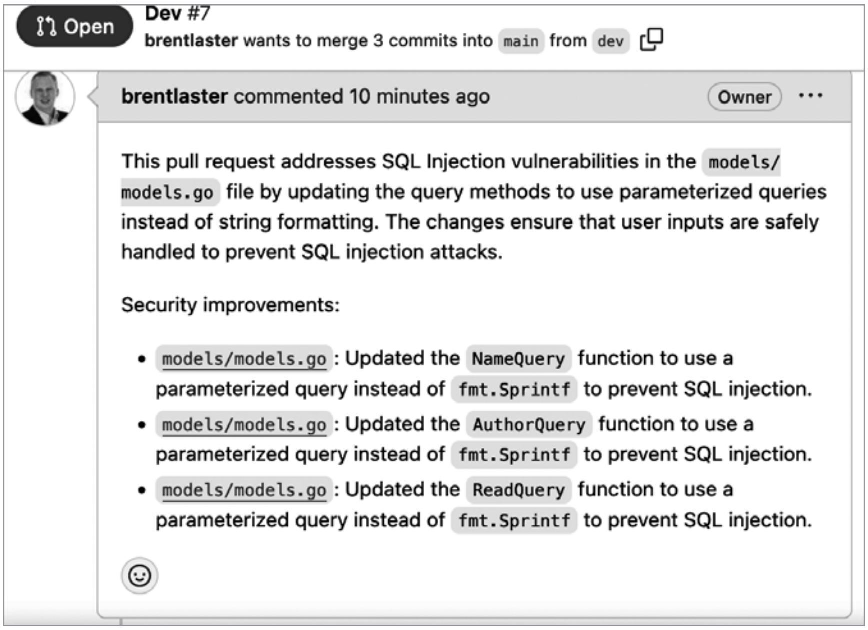
Add a description

Add your description

Summary

Generate a summary of the changes in this pull request.

Рис. 9.21. Pull-запрос, сгенерированный по ссылке



Dev #7

brentlaster wants to merge 3 commits into `main` from `dev`

brentlaster commented 10 minutes ago

This pull request addresses SQL Injection vulnerabilities in the `models/models.go` file by updating the query methods to use parameterized queries instead of string formatting. The changes ensure that user inputs are safely handled to prevent SQL injection attacks.

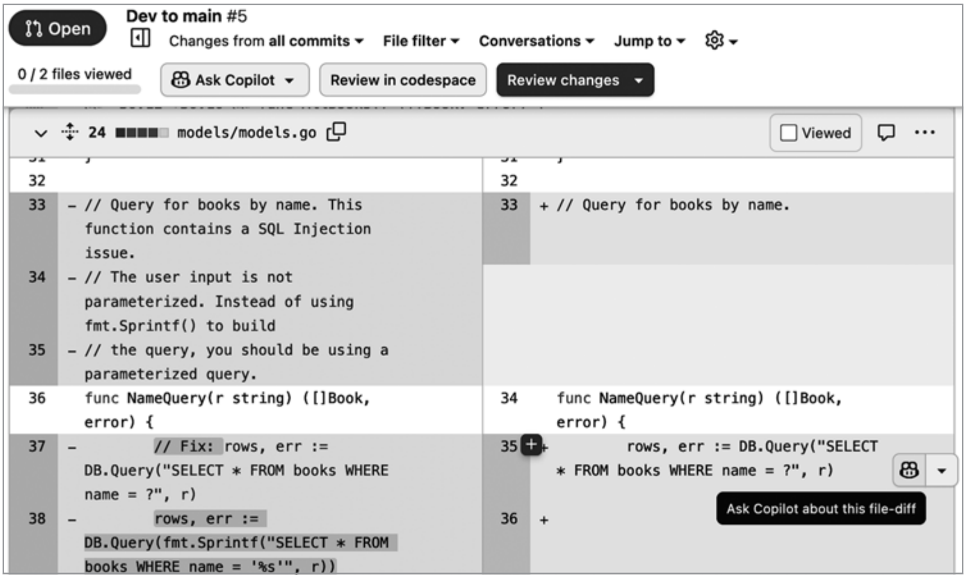
Security improvements:

- `models/models.go`: Updated the `NameQuery` function to use a parameterized query instead of `fmt.Sprintf` to prevent SQL injection.
- `models/models.go`: Updated the `AuthorQuery` function to use a parameterized query instead of `fmt.Sprintf` to prevent SQL injection.
- `models/models.go`: Updated the `ReadQuery` function to use a parameterized query instead of `fmt.Sprintf` to prevent SQL injection.

Рис. 9.22. Описание pull-запроса, сгенерированное Copilot

Анализ изменений в коде с помощью Copilot

В pull-запросе, с которым мы работали, Copilot предоставил ссылки на изменения между двумя ветками. Если вы хотите более глубоко проанализировать одно исправление или набор изменений в файле, то можете перейти по этим ссылкам или сразу на вкладку Files changed, чтобы просмотреть различия (рис. 9.23).



```
- // Query for books by name. This function contains a SQL Injection issue.
- // The user input is not parameterized. Instead of using fmt.Sprintf() to build
- // the query, you should be using a parameterized query.
  func NameQuery(r string) ([]Book, error) {
-         // Fix: rows, err := DB.Query("SELECT * FROM books WHERE name = ?", r)
-         rows, err := DB.Query(fmt.Sprintf("SELECT * FROM books WHERE name = '%s'", r))
```

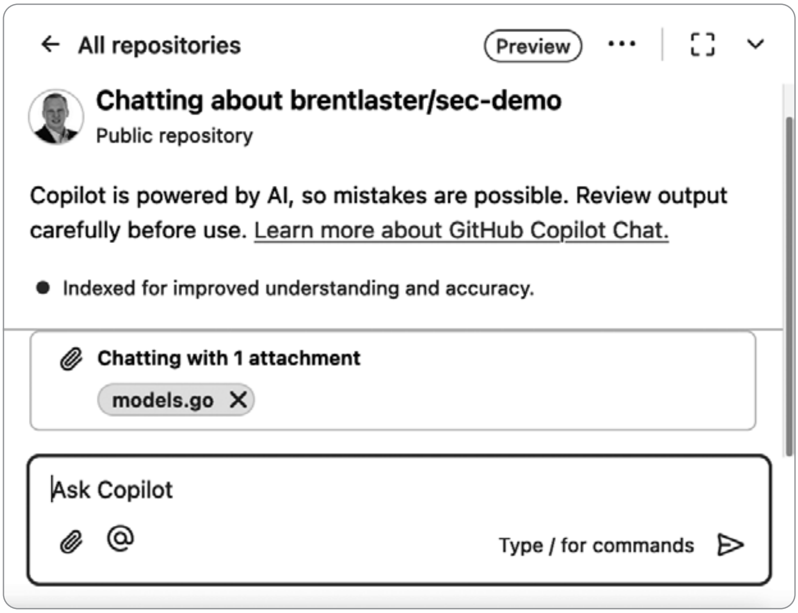
```
+ // Query for books by name.
  func NameQuery(r string) ([]Book, error) {
+         rows, err := DB.Query("SELECT * FROM books WHERE name = ?", r)
+
```

Рис. 9.23. Файлы, измененные с помощью pull-запроса

На этой вкладке pull-запроса находятся две кнопки для открытия Copilot. Внизу справа — стандартная кнопка для открытия интерфейса Copilot Chat. При нажатии кнопки наверху Ask Copilot (рядом с кнопкой Review changes) текущий файл автоматически прикрепляется к контексту чата и открывается диалоговое окно чата (рис. 9.24).

Добавление файла в качестве контекста настраивает систему так, что ответы на все последующие промпты в этом диалоговом окне будут формироваться на основе файла `models.go`. Например, в этом окне можно попросить: «Выдай краткое описание изменений в этом файле», и Copilot предоставит информацию о файле, использованном как контекст (рис. 9.25).

Кроме того, есть третий (иногда скрытый) способ доступа к Copilot при подобном сравнении различий в файлах. Прокрутив экран вверх, к началу выбранного различия, вы увидите маленький плавающий значок Copilot со стрелкой, направленной вниз. При щелчке на стрелке открывается список с двумя командами: Explain и Attach to current thread (рис. 9.26).



Chatting about brentlaster/sec-demo

Public repository

Copilot is powered by AI, so mistakes are possible. Review output carefully before use. Learn more about GitHub Copilot Chat.

● Indexed for improved understanding and accuracy.

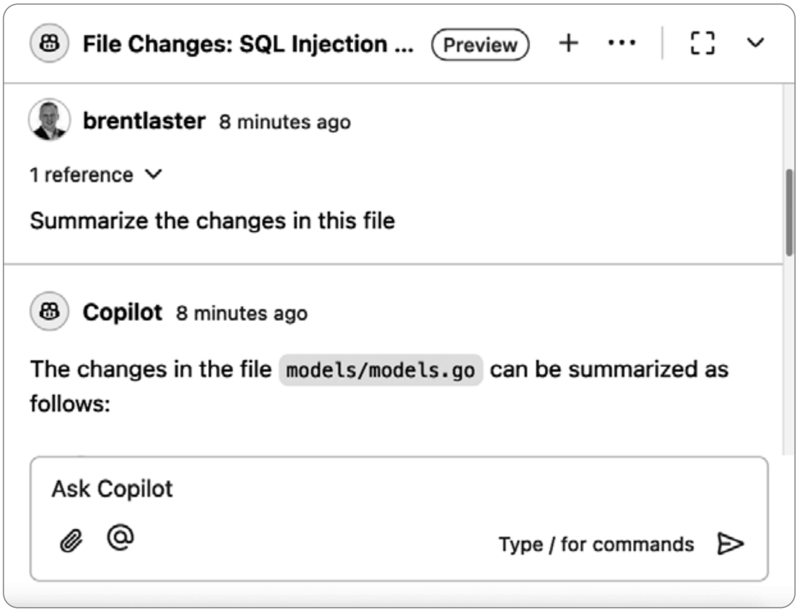
Chatting with 1 attachment

models.go ✕

Ask Copilot

Type / for commands

Рис. 9.24. Диалоговое окно Ask Copilot



brentlaster 8 minutes ago

1 reference ⌄

Summarize the changes in this file

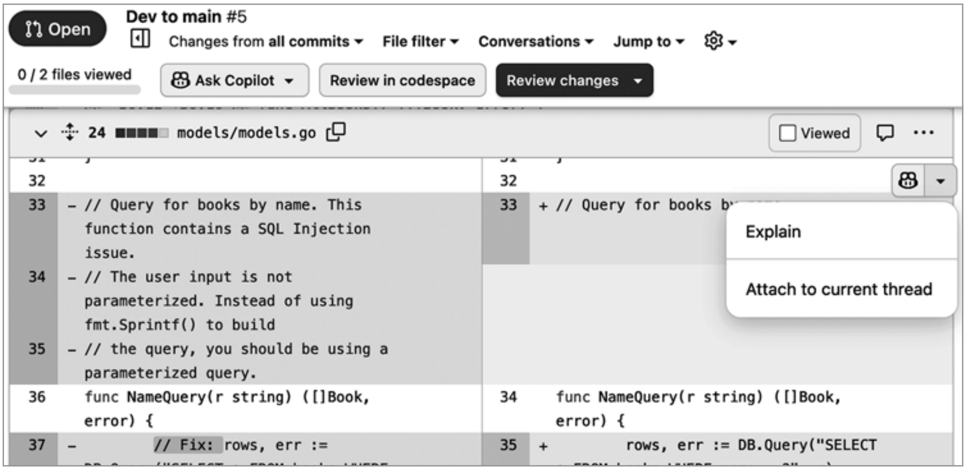
Copilot 8 minutes ago

The changes in the file `models/models.go` can be summarized as follows:

Ask Copilot

Type / for commands

Рис. 9.25. Информация об изменениях в файле из Ask Copilot



```
32                                              32
33   - // Query for books by name. This         33   + // Query for books by
       function contains a SQL Injection
       issue.
34   - // The user input is not
       parameterized. Instead of using
       fmt.Sprintf() to build
35   - // the query, you should be using a
       parameterized query.
36     func NameQuery(r string) ([]Book,        34     func NameQuery(r string) ([]Book,
       error) {                                        error) {
37   -         // Fix: rows, err :=             35   +         rows, err := DB.Query("SELECT
```

Рис. 9.26. Дополнительные команды Copilot, доступные при работе с различиями

Команда Explain делает примерно то же, что и функциональность объяснения Copilot, с которой мы работали ранее. Команда Attach to current thread позволяет задать конкретное различие как контекст для чата Copilot. Например, если выбрать эту команду для различия, показанного на рис. 9.26, то, открыв диалоговое окно чата для Copilot в следующий раз, вы увидите, что в качестве контекста указано одно изменение, начинающееся со строки 33, а не весь файл (рис. 9.27).

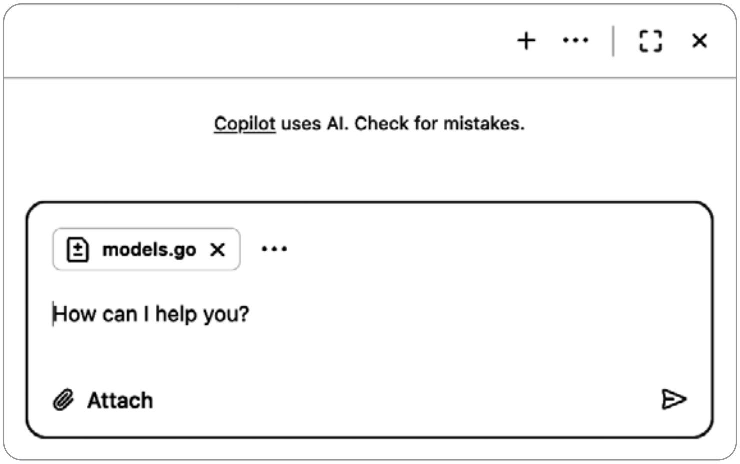
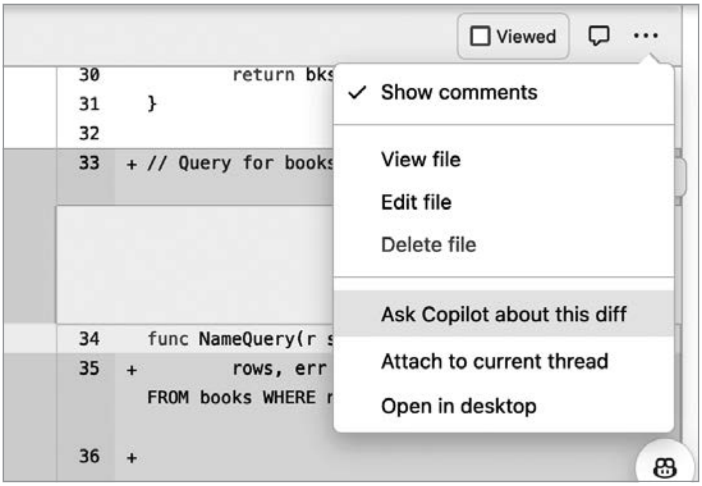

Copilot uses AI. Check for mistakes.

Рис. 9.27. Контекст, привязанный к различию

Обратите внимание: если вы щелкнете на меню Conversation (...) в правом верхнем углу окна различий, то откроется меню с дополнительной функциональностью Copilot (рис. 9.28).



```
        return bks
}

+ // Query for books
func NameQuery(r s
+        rows, err
  FROM books WHERE r

+
```

Рис. 9.28. Обращение к Copilot из меню Conversation

Оба способа позволяют добавить файл в контекст чата, как и кнопка Ask Copilot.

В завершение главы рассмотрим последнюю тему: как Copilot может помочь с задачами, или проблемами, GitHub.

Использование Copilot с задачами GitHub

В Copilot есть встроенная функциональность для обобщения задач (часто также называют проблемами) GitHub Issues. Для их анализа используются различные инструменты (такие как собственные API GitHub). В ходе анализа выделяется тело задачи, заголовок, комментарии и даже связанные pull-запросы (если они есть). Цель — создать сводку основных моментов, которые необходимо учитывать. В зависимости от объема информации, доступной по задаче, в сводке обычно приводится описание задачи, компонентов, на которые она влияет, требуемой функциональности и возможные способы исправления.

Такие сводки полезны по нескольким причинам.

Экономия времени

Сводка помогает разработчику быстро понять основные моменты задачи, не тратя много времени на долгие обсуждения.

Улучшение коммуникаций

Сводка упрощает отслеживание сопутствующих задач или обновлений на всех уровнях.

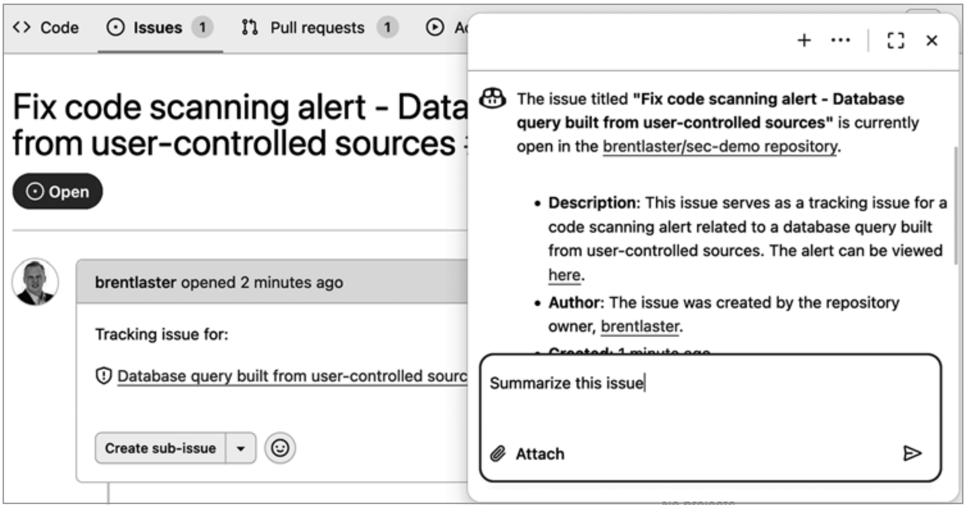
Эффективная приоритизация

Изучив сводку, разработчик может определить приоритетность задач GitHub, уровень их влияния на другие компоненты, а также сложность их решения.

Контроль точности

Как и в случае с результатами, выданными ИИ, за точность сводной информации в конечном счете отвечает пользователь. При формировании сводки задач Copilot может упустить какие-то нюансы и/или подробности, которые могут оказаться важными.

Вернемся к проблеме, связанной с уязвимостью безопасности запросов базы данных, о которой мы говорили в предыдущем разделе. Можно вызвать экран чата GitHub Copilot и ввести промпты для получения информации или помощи с задачей. На рис. 9.29 показан пример, в котором Copilot получает указание вывести сводное описание проблемы.



Fix code scanning alert - Data
from user-controlled sources

brentlaster opened 2 minutes ago

Tracking issue for:

Database query built from user-controlled sourc

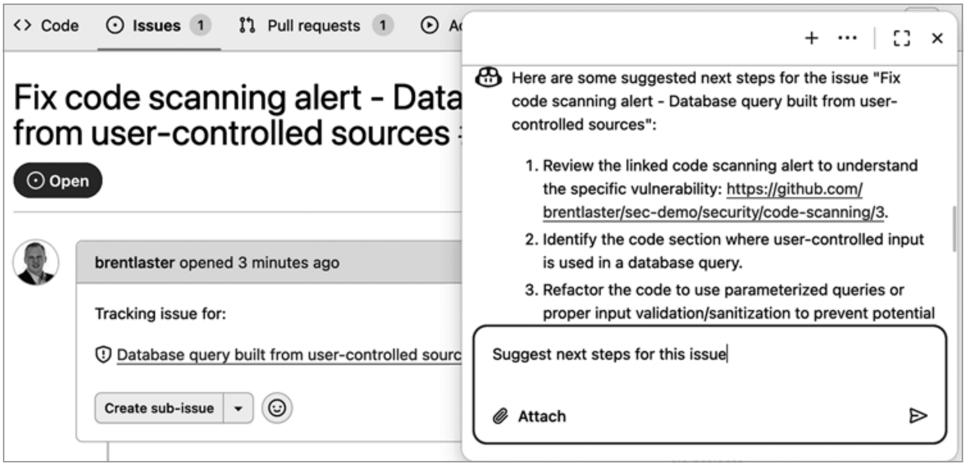
The issue titled **"Fix code scanning alert - Database query built from user-controlled sources"** is currently open in the brentlaster/sec-demo repository.

- **Description:** This issue serves as a tracking issue for a code scanning alert related to a database query built from user-controlled sources. The alert can be viewed here.
- **Author:** The issue was created by the repository owner, brentlaster.

Summarize this issue

Рис. 9.29. Copilot получает указание составить сводное описание проблемы

Можно запросить у Copilot более подробные сведения о задаче, спросить о ее ключевых моментах или даже попросить представить варианты дальнейших действий. На рис. 9.30 показан пример запроса: «Предложи следующие действия для этой задачи».



Fix code scanning alert - Data
from user-controlled sources

brentlaster opened 3 minutes ago

Tracking issue for:

Database query built from user-controlled sourc

Here are some suggested next steps for the issue "Fix code scanning alert - Database query built from user-controlled sources":

1. Review the linked code scanning alert to understand the specific vulnerability: https://github.com/brentlaster/sec-demo/security/code-scanning/3.
2. Identify the code section where user-controlled input is used in a database query.
3. Refactor the code to use parameterized queries or proper input validation/sanitization to prevent potential

Suggest next steps for this issue

Рис. 9.30. Copilot предлагает дальнейшие действия для решения задачи

Генеративный ИИ недетерминирован, поэтому качество ответа может значительно варьироваться в плане детализации. В одних случаях вы можете получить более общие инструкции, в других — более подробные.

Резюме

В этой главе мы рассмотрели различные варианты интеграции Copilot с веб-интерфейсом GitHub. Благодаря этой прямой интеграции возможности Copilot значительно расширяются и становятся доступными при работе с репозиториями и основными механизмами GitHub (такими как задачи (issues) и pull-запросы).

Интеграция с GitHub осуществляется в основном через чат-интерфейс Copilot. Репозитории GitHub предоставляют несколько возможностей для вызова диалогового окна чата и общения с Copilot на естественном языке. Если вы работаете над проблемой, pull-запросом или исправлением и запускаете новый сеанс чата, то Copilot идентифицирует его как контекст и предложит соответствующие вопросы в качестве отправной точки.

Чтобы получать более конкретные ответы, адаптированные для одного или нескольких файлов, вы можете добавлять в чат файлы в качестве контекста. Кроме того, при просмотре изменений в pull-запросах вы можете добавить отдельные файлы или даже попросить Copilot при генерировании ответов использовать только часть файла с конкретным изменением.

Copilot также может составить сводку изменений, внесенных в pull-запрос, в виде набора комментариев. Эти комментарии будут содержать ссылки на конкретные изменения, что поможет сократить объем работы по созданию pull-запроса. Вдобавок Copilot может выдать сводку о проблемах GitHub и ответить на вопросы, связанные с этой темой.

Наконец, стоит отметить, что Copilot можно добавить в качестве рецензента pull-запросов. Выступая в роли автоматизированного рецензента, Copilot может предоставить обратную связь и предложения непосредственно в теле pull-запроса так же, как это сделал бы рецензент-человек.

В следующей главе вы узнаете, как расширить функциональность Copilot и интегрировать его с другими инструментами с помощью Copilot Extensions.

ГЛАВА 10

Расширение функциональности Copilot

GitHub Copilot — невероятно полезный инструмент. И его можно сделать еще более эффективным, расширяя его возможности за счет интеграции с другими инструментами, в том числе и теми, которые вы напишете сами. Эти механизмы называются *расширениями*.

Расширения Copilot изменяют функциональность GitHub Copilot, адаптируя ее к конкретным сценариям использования или средам. Например, расширения могут интегрировать пользовательские или сторонние приложения с Copilot через чат-интерфейс. Так, можно установить расширение Copilot Docker (https://oreil.ly/QS54S), а затем задавать в чате вопросы наподобие «@Docker, как мне контейнеризировать этот проект?».

В этой главе вы узнаете, что такое расширения GitHub Copilot, как находить общедоступные расширения и использовать их. Затем мы перейдем к более подробному описанию различных типов расширений, принципов их работы, а также их достоинств и недостатков. Кроме того, мы рассмотрим некоторые нюансы на простых, но полезных примерах реализации. Но прежде, чем переходить к этим темам, стоит прояснить некоторые неоднозначные термины.

Расширения Copilot и VS Code Extensions для Copilot

Расширения функциональности Copilot могут быть реализованы одним из двух способов. Они могут быть спроектированы для работы на любой платформе, на которой поддерживается Copilot Chat (IDE, GitHub.com и т. д.), независимо от нее. Или могут создаваться для среды VS Code, чтобы использовать ее функции и работать только на этой платформе (либо в другой IDE в случае портирования).

В этой главе я буду использовать термины «расширения VS Code для Copilot» или «участники чата» для обозначения расширений, реализованных через VS Code. Для *нативных* расширений Copilot (реализованных для независимости от платформы) будет использоваться более короткое название «расширения Copilot».

В обоих случаях получить доступ к функциональности расширения можно через чат-интерфейс Copilot. Расширения VS Code для Copilot просто используют функциональность среды. Расширения Copilot требуют отдельно настроенного и специализированного приложения GitHub App, связывающего чат-интерфейс с бэкенд-кодом.

Подробности реализации и конфигурации обоих типов расширений мы рассмотрим позднее в этой главе. В частности, вы узнаете, что такое приложения GitHub App и как их можно использовать с расширениями Copilot. Но сначала мы более подробно поговорим о том, что собой представляют эти расширения в целом.

Серверные и клиентские расширения

Расширения Copilot удобно рассматривать как серверные — в том смысле, что им для работы необходимо связываться с приложением, работающим на стороне GitHub (хотя это не официальное описание). В свою очередь, расширения VS Code для Copilot можно рассматривать как клиентские, поскольку они работают, используя API клиента VS Code, на котором установлены.

Что такое Copilot Extensions

В основе пользовательских функций Copilot лежит фреймворк, включающий в себя набор как API, так и низкоуровневых системных возможностей. Его могут использовать третьи стороны и пользователи для расширения функционала Copilot. Кроме того, другие приложения с его помощью могут предоставлять свои услуги через Copilot Chat напрямую пользователю. Вы как пользователь получаете возможность вызывать другие инструменты и взаимодействовать с ними по аналогии с тем, как вы взаимодействуете с ИИ Copilot в его исходном состоянии.

Как указано в документации Copilot (https://oreil.ly/B7yAU), расширения используются во многих ситуациях.

Поиск в документации

Copilot Chat получает возможность обратиться с запросом к стороннему сервису документации, чтобы найти информацию по конкретной теме.

Программирование с помощью ИИ

Сторонняя модель ИИ используется для получения предложений о коде.

Выборка данных

Расширение Copilot может позволить Copilot Chat обратиться к стороннему сервису данных за информацией о конкретной теме.

Выполнение действий

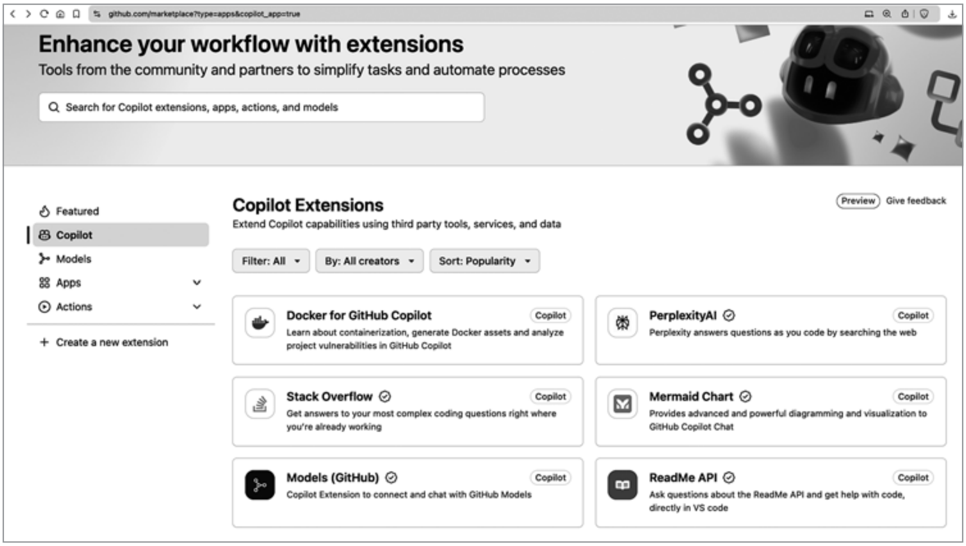
Copilot Chat получает возможность выполнять определенные действия — например, отправлять сообщение на доску объявлений или обновлять элемент отслеживания во внешней системе.

Используя расширения Copilot, вы словно предоставляете Copilot Chat дополнительную информацию о другом приложении или наборе данных и добавляете специализированные навыки, которыми сможете воспользоваться при необходимости.

Расширения Copilot создаются для обеспечения кросс-платформенной совместимости, управления приложением и поддержки. Они работают везде, где работает чат. Они могут быть приватными или общедоступными, а при желании их можно опубликовать на GitHub Marketplace (маркет расширений и улучшений Copilot). И это хорошее место, чтобы начать изучать расширения, в том числе общедоступные.

Загрузка Copilot Extensions из Marketplace

Вы можете создавать собственные расширения (как будет показано далее в этой главе), а организации и компании могут создавать *приватные*. Кроме того, существует немало готовых *общедоступных* расширений от сторонних компаний и организаций. Чтобы ознакомиться с ними, зайдите на GitHub Marketplace (https://github.com/marketplace) и в левом столбце выберите пункт Copilot. На рис. 10.1 показан главный экран расширений Copilot в Marketplace.



Enhance your workflow with extensions

Tools from the community and partners to simplify tasks and automate processes

Copilot Extensions

Extend Copilot capabilities using third party tools, services, and data

Docker for GitHub Copilot
Learn about containerization, generate Docker assets and analyze project vulnerabilities in GitHub Copilot

PerplexityAI
Perplexity answers questions as you code by searching the web

Stack Overflow
Get answers to your most complex coding questions right where you're already working

Mermaid Chart
Provides advanced and powerful diagramming and visualization to GitHub Copilot Chat

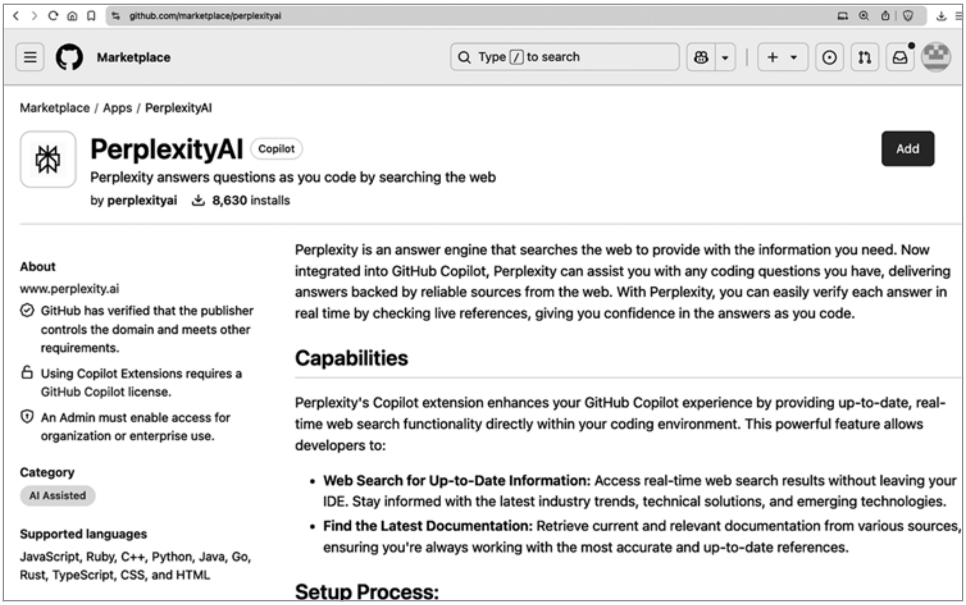
Models (GitHub)
Copilot Extension to connect and chat with GitHub Models

ReadMe API
Ask questions about the ReadMe API and get help with code, directly in VS code

Рис. 10.1. Расширения Copilot на Marketplace

Для примера выберите расширение PerplexityAI, позволяющее собирать информацию через веб-поиск в режиме реального времени. При щелчке на ссылке в Marketplace открывается страница с информацией о том, что собой представляет расширение и что оно может делать (рис. 10.2).

Если вы нажмете кнопку Add (наверху справа) или прокрутите страницу вниз, то попадете в раздел страницы, откуда можно установить расширение (рис. 10.3). Расширение PerplexityAI можно скачать бесплатно.



PerplexityAI

Perplexity answers questions as you code by searching the web

by perplexityai 8,630 installs

About

www.perplexity.ai

GitHub has verified that the publisher controls the domain and meets other requirements.

Using Copilot Extensions requires a GitHub Copilot license.

An Admin must enable access for organization or enterprise use.

Category

AI Assisted

Supported languages

JavaScript, Ruby, C++, Python, Java, Go, Rust, TypeScript, CSS, and HTML

Perplexity is an answer engine that searches the web to provide with the information you need. Now integrated into GitHub Copilot, Perplexity can assist you with any coding questions you have, delivering answers backed by reliable sources from the web. With Perplexity, you can easily verify each answer in real time by checking live references, giving you confidence in the answers as you code.

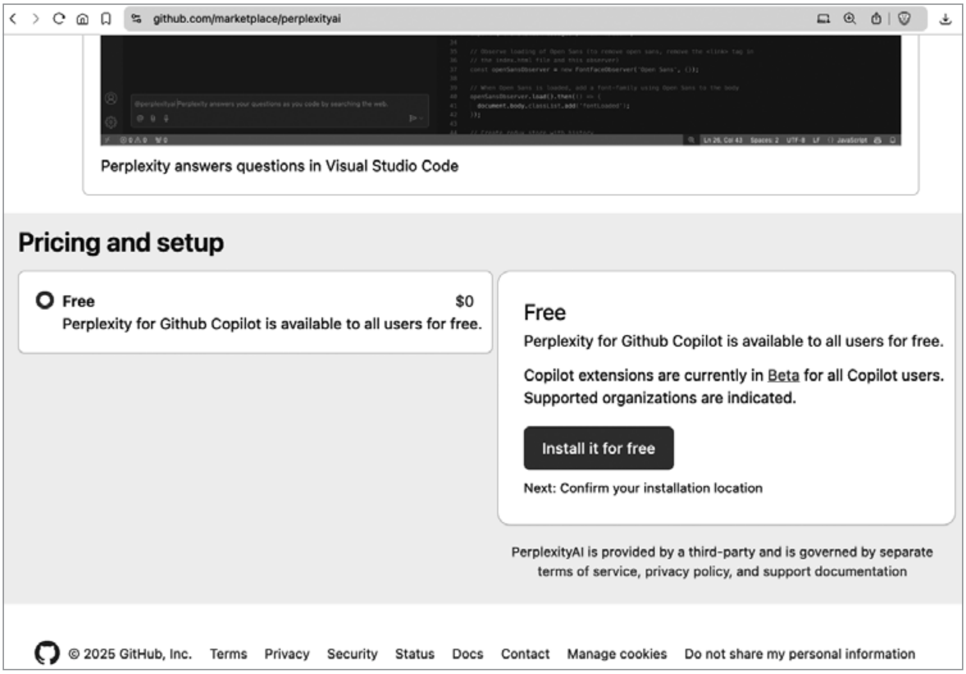
Capabilities

Perplexity's Copilot extension enhances your GitHub Copilot experience by providing up-to-date, real-time web search functionality directly within your coding environment. This powerful feature allows developers to:

- **Web Search for Up-to-Date Information:** Access real-time web search results without leaving your IDE. Stay informed with the latest industry trends, technical solutions, and emerging technologies.
- **Find the Latest Documentation:** Retrieve current and relevant documentation from various sources, ensuring you're always working with the most accurate and up-to-date references.

Setup Process:

Рис. 10.2. Главная страница расширения



Perplexity answers questions in Visual Studio Code

Pricing and setup

○ **Free** $0

Perplexity for Github Copilot is available to all users for free.

Free

Perplexity for Github Copilot is available to all users for free.

Copilot extensions are currently in Beta for all Copilot users. Supported organizations are indicated.

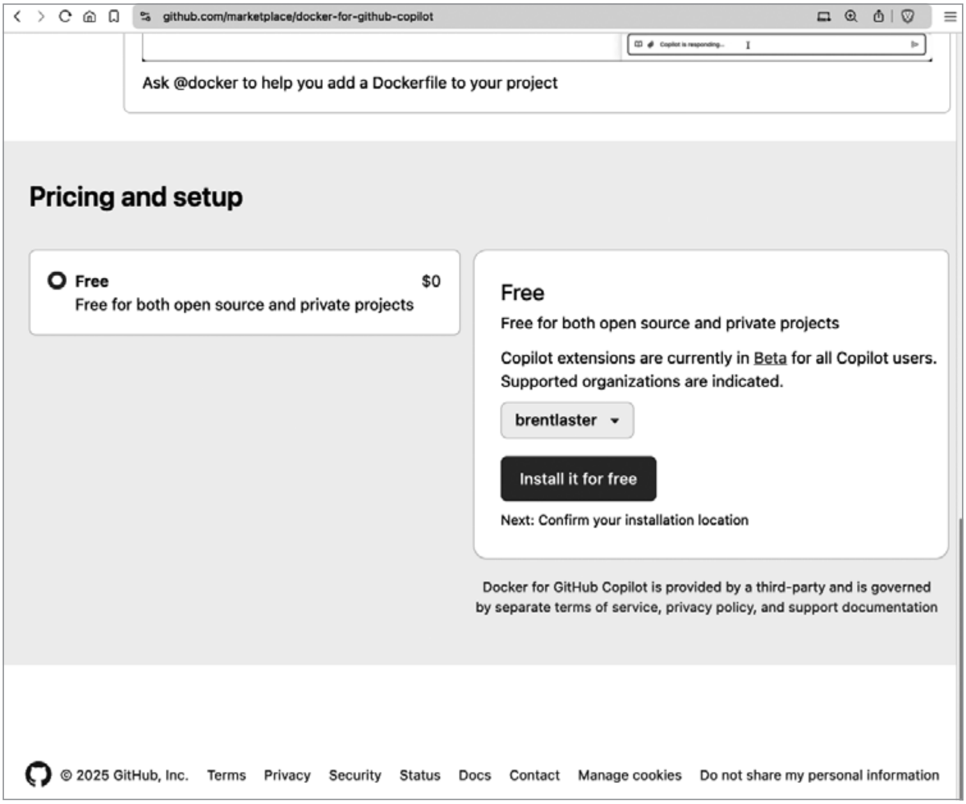
Next: Confirm your installation location

PerplexityAI is provided by a third-party and is governed by separate terms of service, privacy policy, and support documentation

© 2025 GitHub, Inc.

Рис. 10.3. Страница установки расширения

Расширения могут потребовать выбрать пользователя или организацию, которым будет разрешено использовать расширение. Чтобы увидеть подобный пример, выберите установку расширения Docker (рис. 10.4). В свою очередь, расширение PerplexityAI предназначено для общего поиска, поэтому может использоваться для всех репозиториев, связанных с пользователем, который его установил.



Ask @docker to help you add a Dockerfile to your project

Pricing and setup

○ Free $0
Free for both open source and private projects

Free
Free for both open source and private projects

Copilot extensions are currently in Beta for all Copilot users. Supported organizations are indicated.

Next: Confirm your installation location

Docker for GitHub Copilot is provided by a third-party and is governed by separate terms of service, privacy policy, and support documentation

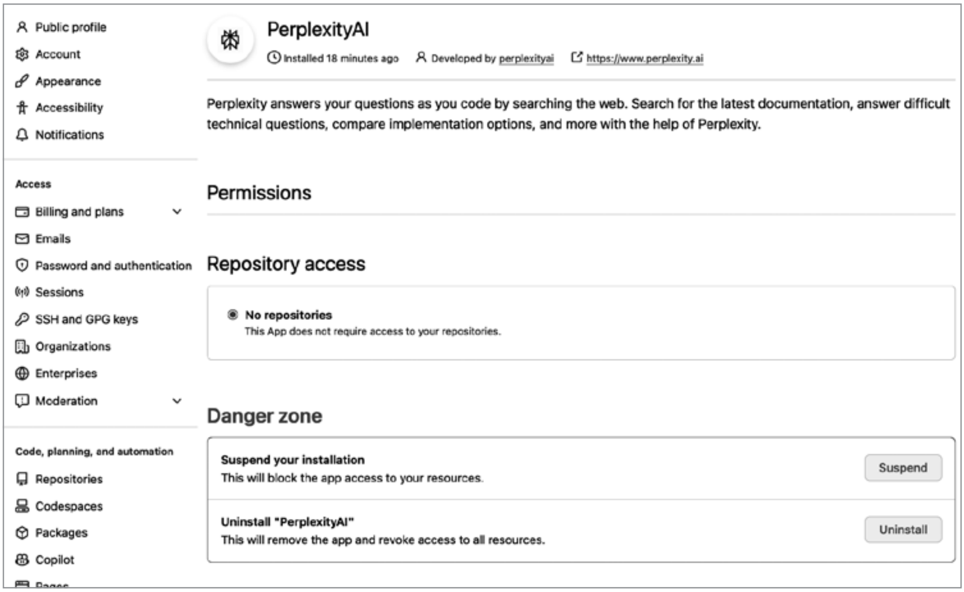
© 2025 GitHub, Inc.

Рис. 10.4. Установка расширения Docker с выбором пользователя или организации

Большинство расширений Copilot получают доступ к репозиториям и вашей учетной записи через приложения GitHub App. По сути, это бэкенд-приложения, которые вы устанавливаете в вашей учетной записи GitHub и даете им разрешение (авторизуете) на работу с вашими репозиториями. Для расширений Copilot, использующих приложение, оно служит мостом между чат-интерфейсом и кодом, реализующим функциональность расширения. Приложение предоставляет доступ и управляющую прослойку между Copilot и операциями расширения.

Когда расширение будет установлено, вы сможете увидеть процесс интеграции приложения. Вам будет предложено авторизовать связанное приложение

GitHub App в вашей учетной записи, пройдя аутентификацию на GitHub. После авторизации открывается страница конфигурации установленного приложения (рис. 10.5).



PerplexityAI

Installed 18 minutes ago Developed by perplexityai https://www.perplexity.ai

Perplexity answers your questions as you code by searching the web. Search for the latest documentation, answer difficult technical questions, compare implementation options, and more with the help of Perplexity.

Permissions

Repository access

● **No repositories**
This App does not require access to your repositories.

Danger zone

Suspend your installation
This will block the app access to your resources.

Uninstall "PerplexityAI"
This will remove the app and revoke access to all resources.

Рис. 10.5. Конфигурация приложения GitHub App для расширения PerplexityAI

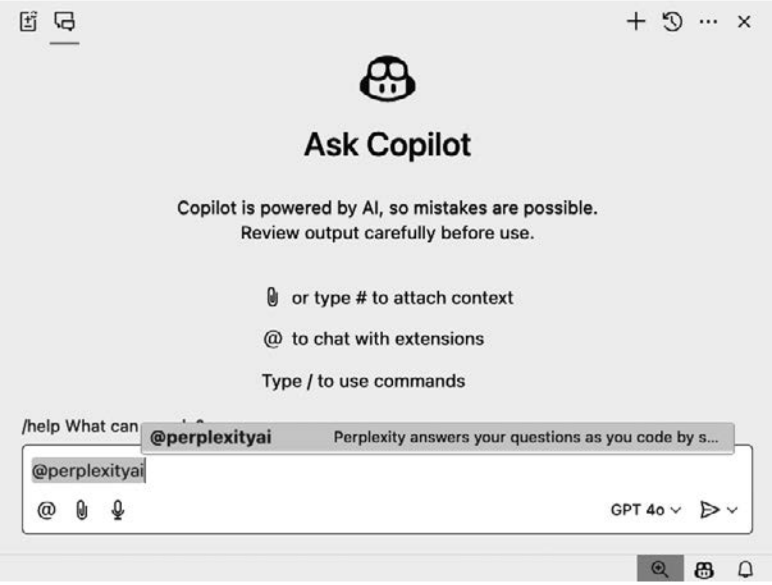
На этой странице можно выбрать разрешения для приложения и/или конкретные репозитории, к которым оно сможет обращаться. В нашем примере приложение для расширения PerplexityAI просто работает со всеми репозиториями и не имеет специальных требований к разрешениям. На этой же странице можно приостановить работу приложения или удалить его.

Изменение конфигурации приложения

Если позднее вам понадобится изменить конфигурацию приложения, а процедура входа уже выполнена, то вы можете перейти в раздел приложений в настройках (https://oreil.ly/WOBZf) и задать нужные изменения там.

После того как фазы установки и авторизации будут завершены, запустите или перезапустите все экземпляры приложений, где активен GitHub Copilot. Затем введите знак `@` и имя расширения в чат-интерфейсе, чтобы запустить новое установленное расширение (рис. 10.6).

Когда вы в первый раз попытаетесь использовать новое расширение через приложение после перезапуска, вам будет предложено авторизовать его: либо для выбранного рабочего пространства, либо для всех рабочих пространств (рис. 10.7).



Ask Copilot

Copilot is powered by AI, so mistakes are possible.
Review output carefully before use.

or type # to attach context

@ to chat with extensions

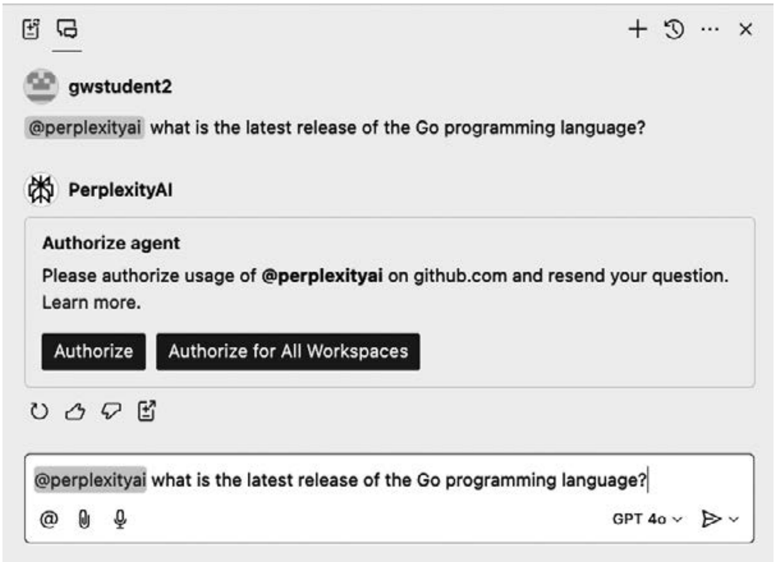
Type / to use commands

/help What can

@perplexityai Perplexity answers your questions as you code by s...

@perplexityai

Рис. 10.6. Вызов приложения PerplexityAI в чате



gwstudent2

@perplexityai what is the latest release of the Go programming language?

PerplexityAI

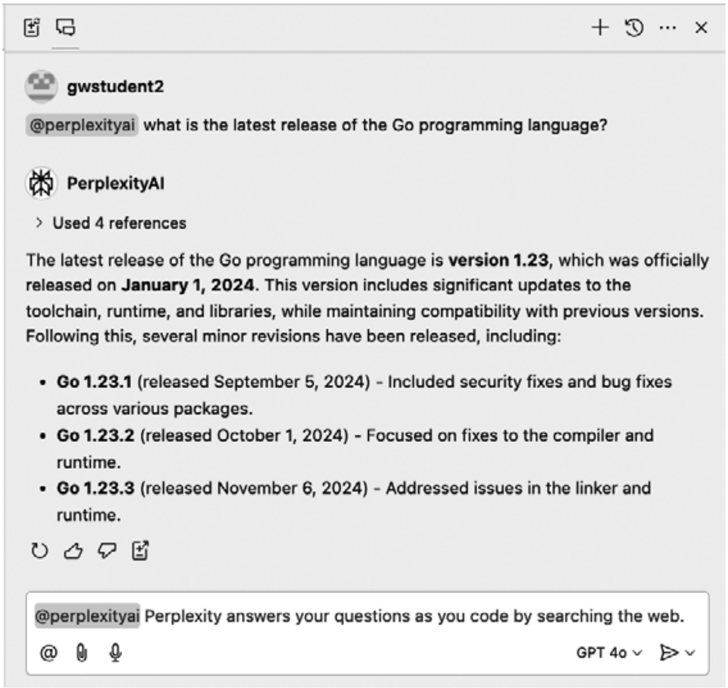
Authorize agent

Please authorize usage of **@perplexityai** on github.com and resend your question.
Learn more.

@perplexityai what is the latest release of the Go programming language?

Рис. 10.7. Авторизация для рабочего пространства

Это делается практически так же, как при исходной авторизации. После того как она будет завершена, вы можете снова ввести свой промпт, а приложение даст ответ (рис. 10.8).


gwstudent2

@perplexityai what is the latest release of the Go programming language?

PerplexityAI

> Used 4 references

The latest release of the Go programming language is **version 1.23**, which was officially released on **January 1, 2024**. This version includes significant updates to the toolchain, runtime, and libraries, while maintaining compatibility with previous versions. Following this, several minor revisions have been released, including:

- **Go 1.23.1** (released September 5, 2024) - Included security fixes and bug fixes across various packages.
- **Go 1.23.2** (released October 1, 2024) - Focused on fixes to the compiler and runtime.
- **Go 1.23.3** (released November 6, 2024) - Addressed issues in the linker and runtime.

@perplexityai Perplexity answers your questions as you code by searching the web.

Рис. 10.8. Ответ PerplexityAI после авторизации

Количество общедоступных расширений Copilot постоянно растет. Если они подходят для вашего приложения или ваших целей, то могут быть полезным вариантом. Но если вам нужна функциональность, которая еще не реализована, или более прямая интеграция с вашими данными или процессами, то, возможно, стоит подумать о разработке и использовании собственного расширения Copilot. Мы поговорим об этом варианте позже, а пока обсудим два возможных типа *реализации* расширений Copilot.

Типы реализации расширений Copilot

Расширения Copilot могут быть реализованы как один из двух типов.

Агент (agent)

Работает как полноценный ИИ-помощник. Это означает, что оно может вести сложные диалоги в чате, выполнять код пользователя и возвращать конкретные ответы.

Набор навыков (skillset)

Может в фоновом режиме вызывать одну или несколько конечных точек API, чтобы получать конкретную информацию или выполнять базовые действия.

Примеры создания обоих типов реализации будут приведены позднее в этой главе. А пока поговорим о преимуществах и недостатках каждого из них.

Агенты — исходный тип расширений Copilot. Они предоставляют наиболее широкую функциональность, если вам нужна сложная или нестандартная обработка и интеграция. Агенты обеспечивают полный контроль, но могут быть весьма сложными в реализации.

Наборы навыков упрощают вызовы других API или выполнение базовых операций. Они проще в реализации, но предоставляемые ими возможности и степень контроля более ограниченны.

Два варианта реализации сравниваются в табл. 10.1.

Таблица 10.1. Сравнение типов расширений Copilot

Параметр	Агенты Copilot	Наборы навыков Copilot
Сложность	Требуют дополнительных усилий по управлению всем процессом взаимодействия с пользователем и генерирования ответов	Предназначены для простой интеграции с минимальными настройками
Уровень контроля	Обеспечивают полный контроль над обработкой запросов и генерированием ответов	Обеспечивают ограниченный контроль над взаимодействиями с пользователем, основное внимание направлено на получение данных и базовые действия
Сценарии применения	Сложные сценарии, где требуется реализовать пользовательскую логику, интегрироваться с другими ИИ-моделями и управлять контекстом беседы	Простые задачи — такие как получение данных из внешнего API или выполнение базовых операций
Доступные интерфейсы	Все (GitHub, IDE)	Все (GitHub, IDE)
Поддержка	GitHub	GitHub

Итак, теперь вы понимаете, чем различаются эти типы, и мы можем рассмотреть их реализацию. Сначала обсудим структурные элементы, из которых собираются расширения Copilot.

Структурные элементы расширений

Структурные элементы расширений — это компоненты, которые нужно собрать вместе так, чтобы расширение стало доступным и нормально работало. При создании расширений Copilot необходимы как минимум следующие компоненты:

- приложение GitHub App с конкретной точкой входа, которая обрабатывает взаимодействие между расширением и Copilot Chat;
- серверный процесс, который может обслуживать бэкенд и предоставлять доступ к конечным точкам;
- бэкенд, который отвечает за функциональность расширения. Эта функциональность может содержать вызовы к LLM, а также интеграцию с внешними инструментами, сервисами и API.

На рис. 10.9 показаны структурные элементы, которые будут использоваться для создания расширений.

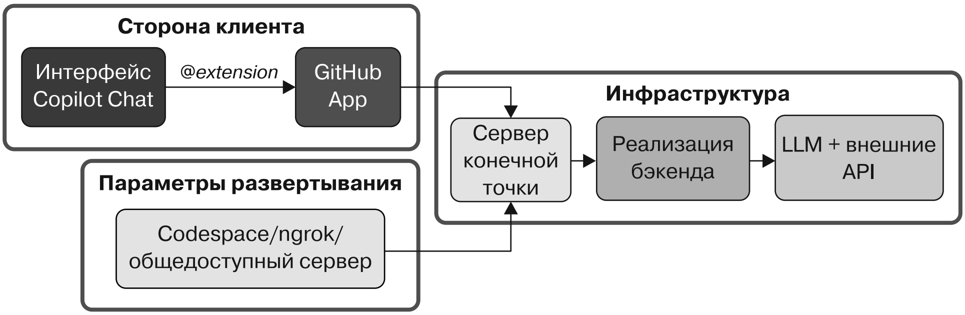


Рис. 10.9. Структурные элементы

Приложения GitHub и серверы конечных точек требуются как для агентов, так и для наборов навыков, но вообще являются универсальными и могут использоваться не только в расширениях. Далее я опишу, как с помощью этих компонентов обеспечивать работу расширений.

Приложения GitHub App

Приложение GitHub App — инструмент, интегрируемый с GitHub для расширения функциональности GitHub.

Такие приложения используются для:

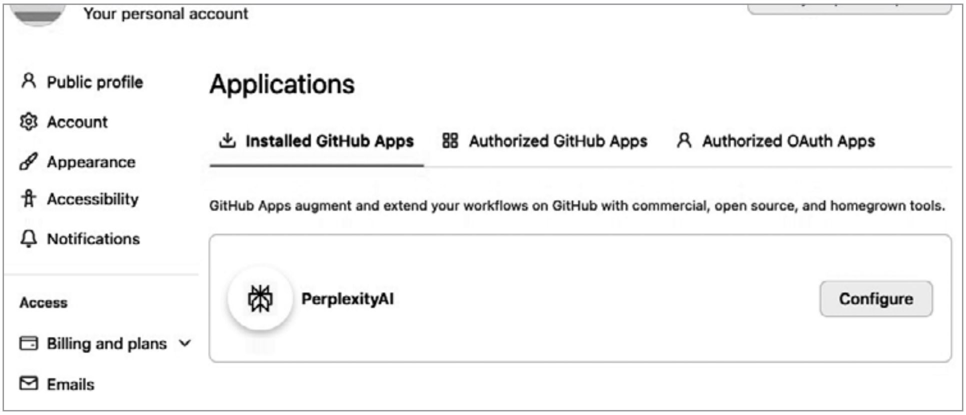
- автоматизации задач;
- интеграции с API и веб-перехватчиками;
- настройки рабочего процесса GitHub.

После создания, установки и регистрации приложение GitHub App может работать независимо от пользователя как авторизованный процесс для выполнения операций, указанных в перечне выше. Его этапы: регистрация приложения (в том числе настройка конфигурации вручную, если потребуется), написание необходимого кода для функциональности и последующая организация запуска через веб-перехватчик, URL обратного вызова и т. д. Кроме того, приложение служит шлюзом для получения авторизации доступа к ресурсам.

URL обратного вызова

В контексте расширений GitHub Copilot термином «URL обратного вызова» обозначается URL, который ваше расширение Copilot предоставляет GitHub. Этот адрес служит точкой, куда система перенаправляет пользователя после успешной аутентификации через ваше расширение. Он действует как «обратный адрес», куда должен вернуться пользователь после завершения процесса авторизации.

Приложение для расширения PerplexityAI, о котором мы говорили выше, служит примером GitHub App. Установив расширение, перейдите в свой профиль GitHub, а затем в левом меню в разделе Integrations выберите Applications. Вы увидите установленное приложение для этого расширения (рис. 10.10).



GitHub Apps augment and extend your workflows on GitHub with commercial, open source, and homegrown tools.

Рис. 10.10. Приложение PerplexityAI, установленное через расширение

Ссылка для установки приложений

Если вы выполнили вход на GitHub, то доступ к списку установленных приложений можно получить, перейдя на страницу Installations (https://oreil.ly/WOBZf).

Для существующих расширений GitHub, доступных публично или в вашей организации либо компании, приложение устанавливается и настраивается автоматически в процессе установки и авторизации расширения. Для расширений, которые вы реализуете самостоятельно, необходимо создать и настроить новое приложение в рамках общего процесса. Более подробно этапы создания и настройки мы рассмотрим позднее в этой главе.

Далее поговорим о другом важном компоненте расширений — о сервере API.

Сервер API

Если коротко, расширение Copilot в форме агента или набора навыков можно представить как приложение, которое подключается к конечной точке, обрабатывающей взаимодействие между расширением и Copilot Chat. *Конечной точкой* (endpoint, эндпоинт) называется API для кода, который получает входные данные в чате, выполняет какие-то операции, а затем выводит результат обратно в чат. Таким образом, наличие доступной конечной точки — ключевое требование для рабочего процесса.

Если расширение устанавливается с Marketplace или с сайта организации/компании, то сервер уже будет подготовлен и настроен. Если вы создаете собственное расширение, то его нужно будет разместить на общедоступном сервере (например, развернув в облачном сервисе или используя туннелирование). Например, задействуя простые сервисы, такие как Cloudflare (https://oreil.ly/m1eVi) или ngrok (https://ngrok.com), вы можете предоставить общедоступный URL для разработки и тестирования.

Использование обоих этих сервисов требует регистрации и входа. Если же вы используете VS Code или GitHub Codespace для разработки и тестирования, то можете открыть доступ к порту и включить переадресацию портов (https://oreil.ly/jgyG3). Главное, о чем следует помнить при выборе этого метода, что видимость порта нужно изменить с приватной, используемой по умолчанию, на общедоступную.

Приложение и конечная точка предоставляют механизм обмена информации с чат-интерфейсом, но основная работа выполняется в бэкенд-коде, реализующем конечную точку. В оставшейся части этой главы я покажу, как создаются такие реализации. Начнем с создания расширения в форме агента.

Реализация расширения в форме агента

Copilot Agent — пользовательский инструмент, внедренный в расширение Copilot. Агенты дополняют возможности Copilot, позволяя пользователям создавать собственные функции и интегрировать их в интерфейс Copilot Chat.

Режим агента Copilot

В главе 4 мы обсуждали режим агента, в котором Copilot может автоматически создавать код и команды терминала, позволяющие получить желаемый результат. В этой главе мы снова поговорим об агентах, но как об отдельных инструменитах, предназначенных для использования расширениями.

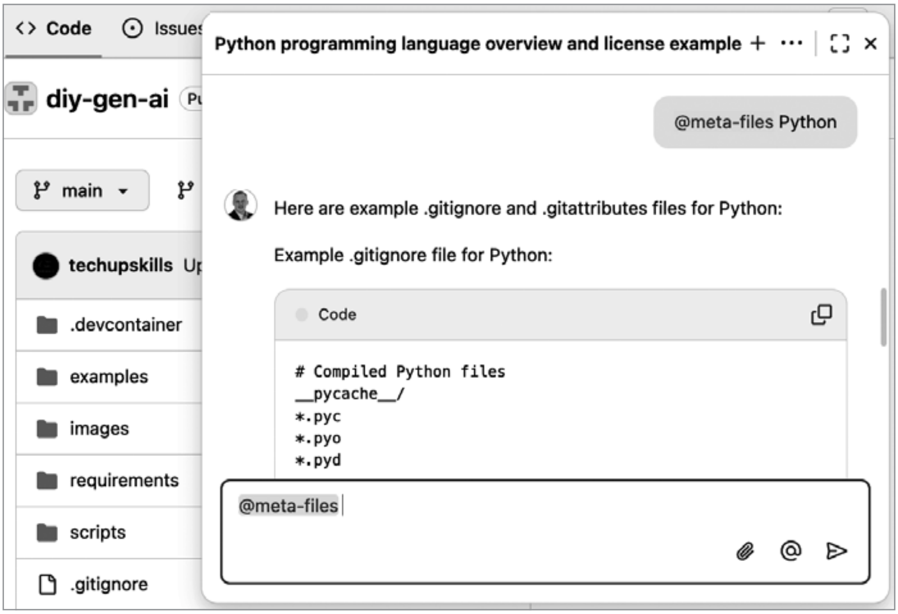
Реализация в форме агента подходит в тех случаях, где вам требуется больший контроль над обработкой запроса или генерированием ответа во время чат-взаимодействия. Агенты могут реализовать пользовательскую логику, интегрироваться с другими LLM и Copilot API, работать с контекстом диалогов в чате и управлять взаимодействием. Но их сложнее создавать и сопровождать.

Базовая реализация

В этом подразделе вы увидите, как реализовать в форме агента простое, но полезное расширение `@meta-files`. Для языка программирования, указанного в промпте, оно сгенерирует файлы `.gitignore` и `.gitattributes`. В качестве альтернативы, если пользователь укажет в промпте название лицензии с открытым исходным кодом, расширение сгенерирует ее примерный текст. Если в лицензии есть места, где нужно указать имя пользователя и текущий год, то расширение попытается подставить соответствующие сведения. Наконец, оно укажет, в каких случаях будет полезен этот тип лицензии.

Для вызова этого расширения использовались промпты `@meta-files Python` (рис. 10.11) и `@meta-files MIT` (рис. 10.12). Код можно найти в репозитории книги на GitHub (https://oreil.ly/93ZY_).

На рис. 10.13 показана общая схема работы расширения в форме агента. Далее мы рассмотрим ее отдельные компоненты более подробно.



@meta-files Python

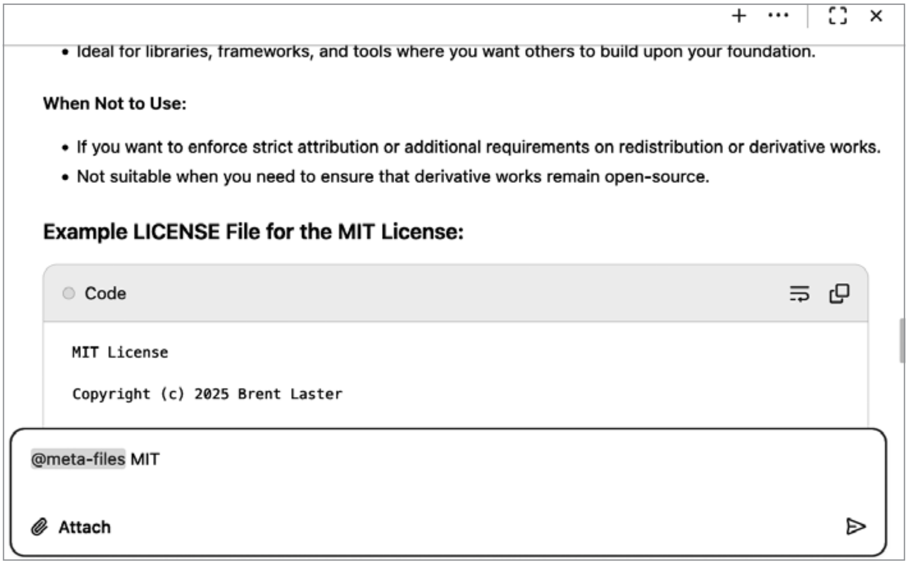
Here are example .gitignore and .gitattributes files for Python:

Example .gitignore file for Python:

```
# Compiled Python files
__pycache__/
*.pyc
*.pyo
*.pyd
```

@meta-files

Рис. 10.11. Расширение генерирует метафайлы для Python



- Ideal for libraries, frameworks, and tools where you want others to build upon your foundation.

When Not to Use:

- If you want to enforce strict attribution or additional requirements on redistribution or derivative works.
- Not suitable when you need to ensure that derivative works remain open-source.

Example LICENSE File for the MIT License:

Code

```
MIT License

Copyright (c) 2025 Brent Laster
```

@meta-files MIT

Рис. 10.12. Расширение генерирует информацию о лицензии

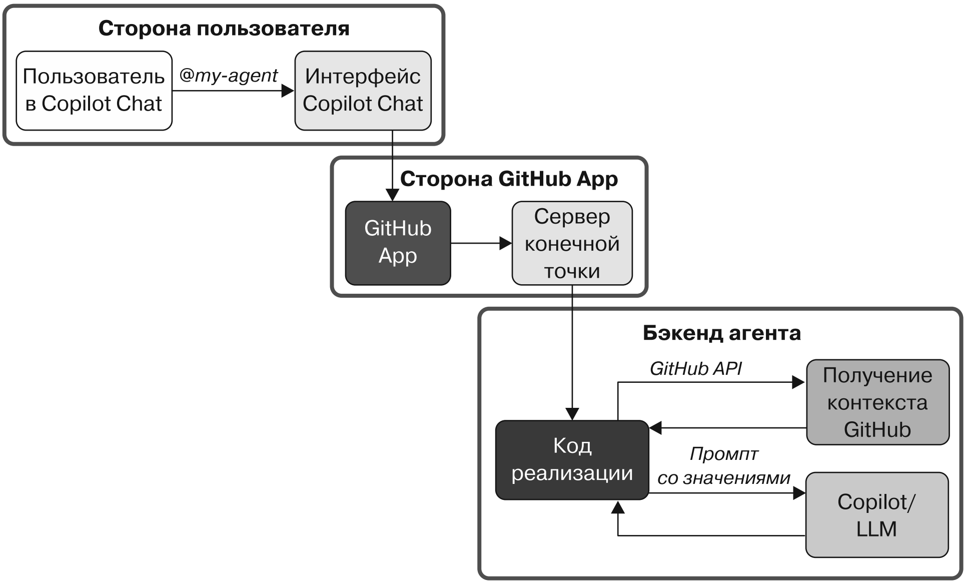


Рис. 10.13. Схема работы расширения в форме агента

Для краткости мы рассмотрим не весь код, а только ключевые фрагменты. Этот пример написан на JavaScript, но вы можете писать расширения практически на любом современном языке программирования.

В этом простом примере мы сосредоточимся на двух стандартных файлах JavaScript: `package.json` и `index.js`. Начнем с содержимого `package.json`:

```
{
  "name": "metafiles-extension",
  "private": "true",
  "description": "Generate metafiles for a GitHub repository",
  "scripts": {
    "start": "node index.js",
    "dev": "node --watch index.js"
  },
  "type": "module",
  "dependencies": {
    "@octokit/core": "^6.1.2",
    "express": "^4.19.2"
  }
}
```

Структура этого кода представляет собой стандартный шаблон для файлов такого типа. В нее можно было добавить несколько других полей, но мы не будем усложнять пример. Единственное, что требует дополнительных пояснений, — зависимости.

Express (https://expressjs.com) — простой фреймворк веб-приложений, помогающий упростить создание веб-приложений и API с Node.js. Создавая расширение-агент, мы фактически создавали веб-приложение с конечной точкой, доступ к которому открывается через GitHub App.

Вторая зависимость, используемая в коде выше, — Octokit (https://github.com/octokit). Это набор библиотек, которые GitHub предоставляет для взаимодействия кода с его REST и GraphQL API. Octokit позволяет программно автоматизировать задачи и управлять репозиториями, а также работать с ресурсами GitHub, такими как pull-запросы и задачи (issues), непосредственно из кода. Здесь он нужен нам, чтобы получить токен доступа и работать с информацией пользователя.

Файл с основной логикой называется `index.js`. Ознакомиться с ним можно на GitHub (https://oreil.ly/GJW8z). Логика в начале файла выполняет необходимое импортирование, запускает приложение Express, публикует приветственное сообщение, а затем переходит к основной обработке.

В ходе обработки Octokit используется для получения токена, а затем начинается стандартный цикл, который выглядит так:

- получение полезной нагрузки, поступающей из чат-интерфейса (через приложение GitHub App);
- дополнение промпта, в котором вы сообщаете ИИ, что вас интересует;
- отправка запроса LLM, чтобы получить ответ от ИИ;
- передача ответа в чат-интерфейс.

Механика процесса довольно проста. Особенно важен этап, когда вы дополняете промпт, чтобы объяснить ИИ, что именно ищете. Код этой части выглядит так:

```
// Определение роли LLM и того, что она должна искать и делать

const messages = payload.messages;

messages.unshift({
  role: "system",
  content: "Найди название языка программирования в сообщении
  от пользователя. Если найдешь, сгенерируй файлы .gitignore
  и .gitattributes для этого языка программирования",
});

messages.unshift({
  role: "system",
  content: "Найди название лицензии ПО с открытым кодом, ее тип
  или сокращение в сообщении от пользователя. Если найдешь,
  то сделай следующее: предоставь несколько ключевых пунктов описания
  лицензии и где она должна/не должна использоваться. Затем сгенерируй
  файл LICENSE для этой лицензии. Если файл содержит текст [NAME]
  или [YOUR NAME], то замени этот текст на " + user.data.name + ".
  Если файл LICENSE содержит текст [YEAR], то замени [YEAR] текущим годом.",
})
```

Этот этап — передача модели подробного промпта с описанием того, на что она должна обратить особое внимание и какой формат вывода нам требуется, — играет ключевую роль в работе расширений в форме агентов. Это похоже на прямое взаимодействие с моделью наподобие ChatGPT. Чем больше информации мы передаем в промпте, тем более качественный ответ можем ожидать от ИИ. Логика, которую мы используем в коде агента, чтобы получить желаемый результат, обычно сводится к созданию хорошего промпта.

Создание хорошего промпта

Существует множество ресурсов с рекомендациями по созданию качественных промптов. В конечном счете лучшим будет тот промпт, который работает так, как нужно вам, с точки зрения анализа правильной информации и возвращения осмысленного ответа.

Один из простых способов создания хорошего промпта выглядит так: сначала вы вводите промпт, а затем дорабатываете его, используя интерфейс Copilot Chat (или ИИ-модель). Когда вы поймете, что промпт вас устраивает, его можно будет добавить в код расширения.

В результате этой базовой подготовки у нас есть основной код, необходимый для расширения. Код для выполнения этого расширения был создан в среде, *отличной от среды эксплуатации*. Перевод расширения в рабочее состояние потребует дополнительной реализации.

GITHUB COPILOT EXTENSIONS SDK

Чтобы упростить доработку реализации для среды эксплуатации, GitHub опубликовал пакет SDK. *GitHub Copilot Extensions SDK*, также известный как *Preview SDK*, — инструмент, предназначенный для упрощения и ускорения реализации расширений GitHub Copilot, написанных в форме агентов. На момент написания книги SDK вышел в альфа-версии и подходил только для расширений, реализованных в форме агентов.

SDK может использоваться для автоматизации разных стандартных задач.

Проверка полезной нагрузки запроса

Гарантирует, что входящие запросы действительны и безопасны.

Разбор полезной нагрузки

Упрощает извлечение релевантных данных из запросов.

Создание ответов

Форматирует ответы в виде, подходящем для Copilot Chat.

Оптимизация взаимодействия с API

Предоставляет средства для интеграции внешних инструментов, API или источников данных непосредственно в Copilot Chat, освобождая разработчика от необходимости писать повторяющийся код интеграции.

Используя SDK для решения подобных шаблонных задач, разработчики могут сосредоточиться на базовой функциональности расширения. Код SDK находится в разделе расширений Copilot на GitHub (https://oreil.ly/Dgdr4). Для краткости (и потому, что SDK еще находится на ранней стадии разработки) я не буду подробно описывать использование функциональности SDK, но базовые примеры распространенных операций можно найти в файле `README.md` проекта (https://oreil.ly/G_IfL). Несколько базовых примеров использования SDK есть и в папке с примерами (https://oreil.ly/PjTdz).

Для завершения реализации необходимы сервер и приложение. Посмотрим, как их создать.

Настройка приложения GitHub App для Copilot Agent Extension

В этом примере мы используем GitHub Codespace для предоставления среды времени выполнения и сервера для конечной точки. Для среды реальной эксплуатации Codespace не подходит, но несколько упростит процесс в рамках нашей базовой демонстрации и тестовых сценариев. Кроме того, при желании вы можете использовать такой сервис, как ngrok.

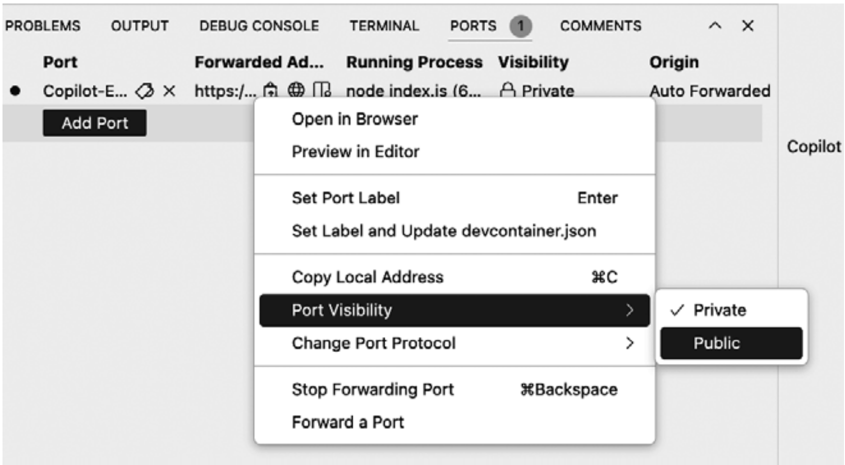
Последовательность действий для создания пространств Codespace описана в документации (https://oreil.ly/DCh4K). После того как пространство заработает, для запуска сервера достаточно двух команд:

```
npm install
npm start
```

Node App и npm

Используемый код представляет собой приложение Node (https://nodejs.org/en/about), а npm (https://npmjs.com) — инструмент Node для его запуска.

На этой стадии у вас будет сервер, выполняемый из пространства Codespace на порте 3000. Прежде чем настраивать GitHub App для расширения, необходимо выполнить несколько действий. Сначала нужно назначить порт, обслуживающий конечную точку, *общедоступным* вместо *приватного*. Для этого перейдите на вкладку Ports в пространстве Codespace, щелкните правой кнопкой мыши на порте в списке и измените значение его атрибута Port Visibility на Public (рис. 10.14).



Port	Forwarded Ad...	Running Process	Visibility	Origin
● Copilot-E...	https:/...	node index.js (6...	Private	Auto Forwarded

Рис. 10.14. Переключение порта в общедоступный режим

Затем в этом же списке скопируйте локальный адрес сервера (рис. 10.15).

Дополнительная информация

Если вы хотите лучше понять, как настроить пространство Codespace, запустить сервер, выбрать порт и т. д., то ознакомьтесь с файлами .devcontainer/devcontainer.json (https://oreil.ly/E9daL) и .vscode/launch.json (https://oreil.ly/HTlM8) в репозитории GitHub.

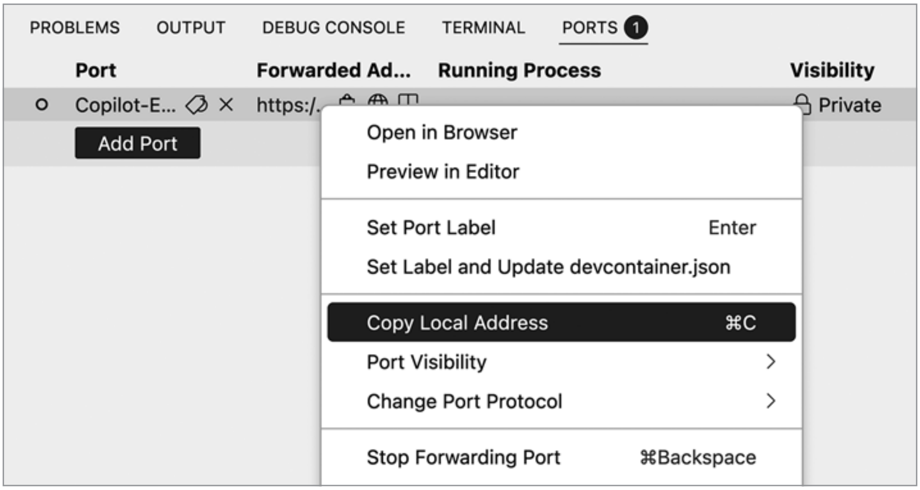


Рис. 10.15. Копирование адреса порта

Реализовав код для агента расширения и запустив сервер, можно переходить к завершению реализации самого расширения и заняться созданием и настройкой соответствующего приложения GitHub App.

Только для целей разработки и тестирования

В примерах этой главы мы создаем код, запускаем простые серверы и реализуем агенты, связанные с конкретным пользователем, только в демонстрационных целях. Чтобы преобразовать эти примеры в экземпляры, готовые к реальной эксплуатации, необходимо добавить функциональность безопасности, хостинга и т. д.

Чтобы создать приложение, войдите в GitHub с тем же идентификатором, который использовался для запуска пространства Codespace. Далее зайдите в настройки разработчика и создайте новое приложение. Сделать это можно, перейдя по ссылке https://github.com/settings/apps/new.

Заполните необходимые поля. В поле App Name должно содержаться уникальное имя. Это имя, которое используется в интерфейсе Copilot Chat для запуска расширения. Например, если вы выберете имя `my-app`, то для вызова расширения в чат-интерфейсе должна использоваться конструкция `@my-app`.

В поле Homepage URL вводится URL для перехода к веб-странице вашего приложения/расширения, если вы ее создали. Если нет, то можете разместить в этом поле ссылку на репозиторий GitHub вашего расширения или даже https://github.com.

В поле Callback URL нужно ввести общедоступный URL с портом, который вы настроили как общедоступный в Codespace. Он необходим для того, чтобы ваше расширение могло отправлять и получать ответы в своем коде. Кроме того, добавьте `/callback` в конец URL.

В наших примерах можно отключить все остальные выбранные параметры на этой странице, например `Webhooks`. Далее нужно настроить видимость приложения. Вы можете сделать его доступным только для себя, пока занимаетесь его разработкой и тестированием. Позднее, когда вы будете готовы поделиться расширением с другими пользователями, вы сможете открыть публичный доступ. На рис. 10.16 показан пример страницы с заполненными полями.


Basic information

GitHub App name *

meta-files

The name of your GitHub App.

Creates useful metafile examples for use in Git repositories

Homepage URL *

https://github.com/learning-gh-copilot/genmeta-ext

The full URL to your GitHub App's website.

Identifying and authorizing users

The full URL to redirect to after a user authorizes an installation.

Read our Callback URL documentation for more information.

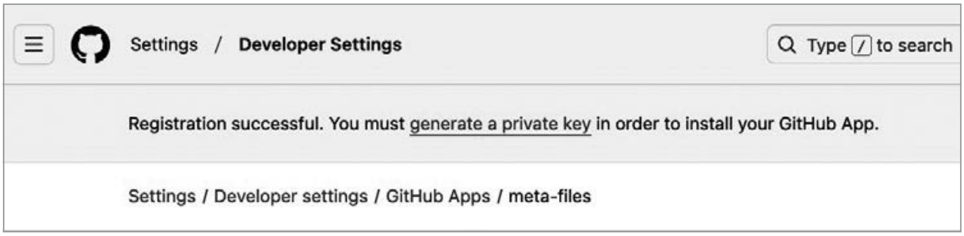
Callback URL

https://probable-halibut-p74p45vjrpc77r6-3000.app.github.dev/callback

Рис. 10.16. Исходная конфигурация расширения в форме агента

Введя данные, нажмите кнопку Create GitHub App, чтобы завершить регистрацию.

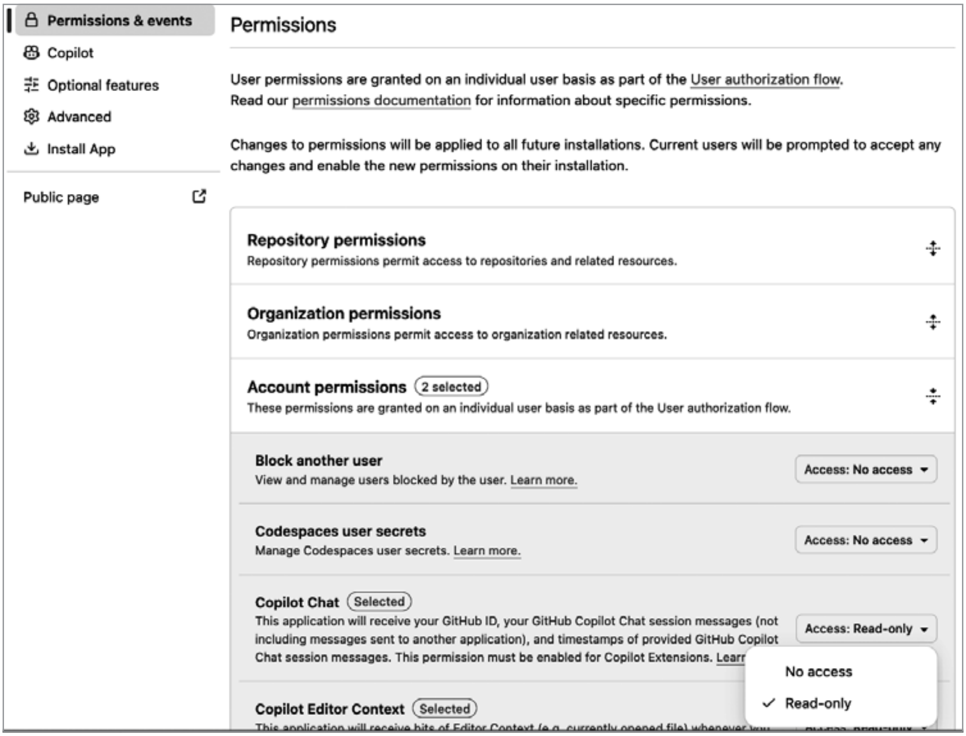
На текущем этапе необходимо сгенерировать приватный ключ, чтобы установить приложение и сертифицировать токены доступа. GitHub предоставит ссылку для генерации и скачивания ключа (рис. 10.17). Он будет автоматически добавлен в конфигурацию приложения.



Registration successful. You must generate a private key in order to install your GitHub App.

Рис. 10.17. Генерирование приватного ключа для установки приложения

Когда приложение создано на странице General, необходимо настроить дополнительные параметры конфигурации, используя разделы в меню слева. На странице Permissions & Events для целей разработки/тестирования необходимо установить только один параметр в разделе Account Permissions. Предоставьте Copilot Chat разрешение только для чтения (рис. 10.18). Сохраните эти изменения.



User permissions are granted on an individual user basis as part of the User authorization flow.
Read our permissions documentation for information about specific permissions.

Changes to permissions will be applied to all future installations. Current users will be prompted to accept any changes and enable the new permissions on their installation.

Repository permissions
Repository permissions permit access to repositories and related resources.

Organization permissions
Organization permissions permit access to organization related resources.

Account permissions (2 selected)
These permissions are granted on an individual user basis as part of the User authorization flow.

Block another user
View and manage users blocked by the user. Learn more.
Access: No access

Codespaces user secrets
Manage Codespaces user secrets. Learn more.
Access: No access

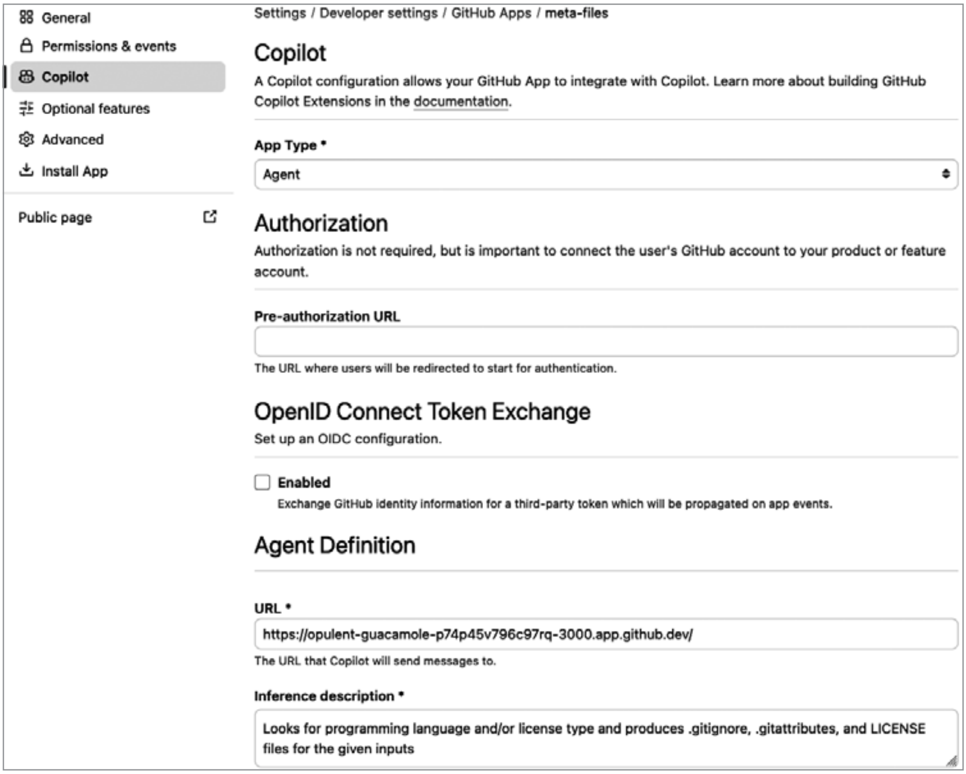
Copilot Chat (Selected)
This application will receive your GitHub ID, your GitHub Copilot Chat session messages (not including messages sent to another application), and timestamps of provided GitHub Copilot Chat session messages. This permission must be enabled for Copilot Extensions. Learr
Access: Read-only

Copilot Editor Context (Selected)

Рис. 10.18. Добавление разрешений для Copilot Chat

На странице специальных настроек для Copilot измените тип приложения с Disabled на Agent (так как мы реализуем расширение в форме агента). Затем в разделе URL вставьте общедоступный URL из своего пространства Codespace.

Наконец, заполните поле Inference description: добавьте краткое описание вашего агента, чтобы другие пользователи смогли понять назначение расширения. Содержимое этого поля отображается при наведении указателя мыши на заголовок расширения в области чата. Заполнив поле, вы сможете сохранить изменения. Заполненная страница показана на рис. 10.19.



Settings / Developer settings / GitHub Apps / **meta-files**

Copilot

A Copilot configuration allows your GitHub App to integrate with Copilot. Learn more about building GitHub Copilot Extensions in the documentation.

App Type *

Agent

Authorization

Authorization is not required, but is important to connect the user's GitHub account to your product or feature account.

Pre-authorization URL

The URL where users will be redirected to start for authentication.

OpenID Connect Token Exchange

Set up an OIDC configuration.

☐ **Enabled**

Exchange GitHub identity information for a third-party token which will be propagated on app events.

Agent Definition

URL *

https://opulent-guacamole-p74p45v796c97rq-3000.app.github.dev/

The URL that Copilot will send messages to.

Inference description *

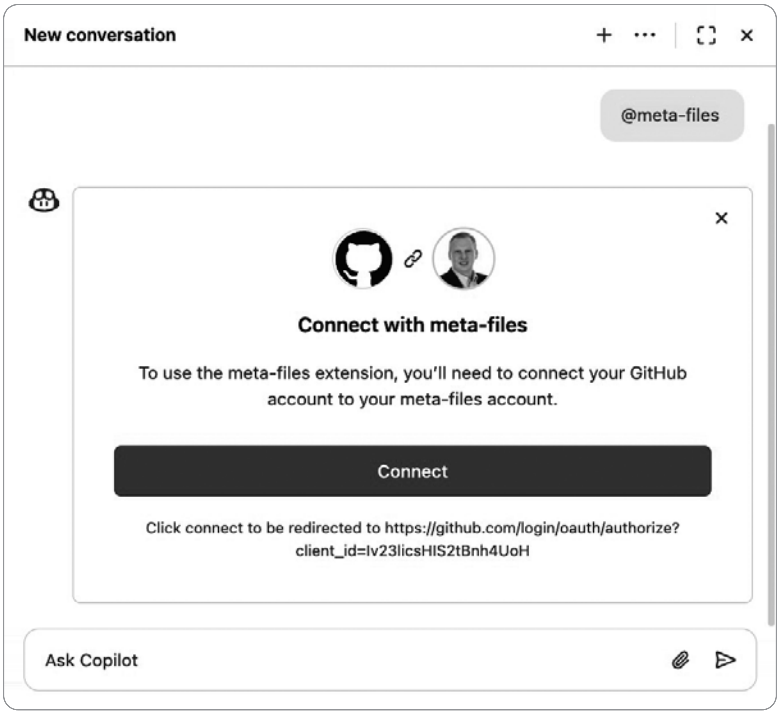
Looks for programming language and/or license type and produces .gitignore, .gitattributes, and LICENSE files for the given inputs

Рис. 10.19. Настройки Copilot для приложения

Это все обязательные поля, которые необходимо заполнить для того, чтобы приложение работало с вашим агентом. На странице Optional Features можно отменить истечение срока действия токенов, а на странице Advanced — задать настройки типа Danger Zone, такие как передача права собственности на приложение, его удаление или публикацию.

Выполнив все описанные выше действия, вы можете предоставить приложению доступ к своей учетной записи для тестирования. Для этого перейдите на страницу Install App и выполните инструкции.

При первой попытке использовать расширение вам нужно будет предоставить доступ к нему (рис. 10.20).



@meta-files

Connect with meta-files

To use the meta-files extension, you'll need to connect your GitHub account to your meta-files account.

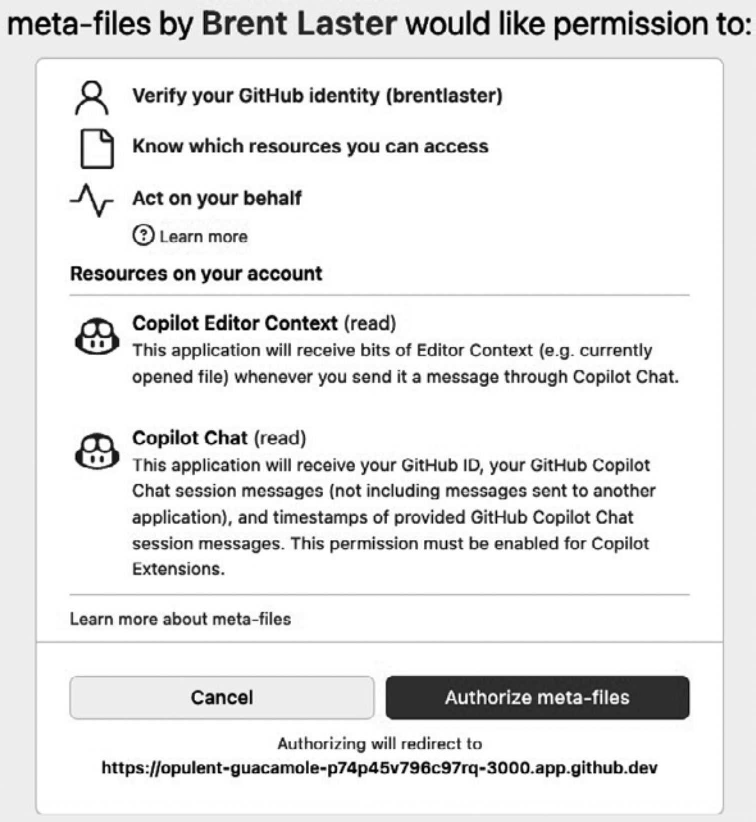
Click connect to be redirected to https://github.com/login/oauth/authorize?client_id=Iv23licsHIS2tBnh4UoH

Рис. 10.20. Подключение приложения при первом использовании

После нажатия кнопки Connect необходимо авторизовать приложение, чтобы дать ему доступ к необходимым ресурсам (рис. 10.21).

Завершив авторизацию приложения, вы готовы к использованию расширения.

Другой способ создания расширений — использование паттерна *набора навыков* (skillset). Этот вариант реализации дает Copilot возможность получать нужную информацию в реальном времени или легко выполнять простую обработку API. Однако помните: эта реализация может быть более простой в плане программирования, но требует более сложной настройки приложения. Расширения, реализованные в форме наборов навыков, мы рассмотрим в следующем разделе.



meta-files by Brent Laster would like permission to:

Verify your GitHub identity (brentlaster)

Know which resources you can access

Act on your behalf

Resources on your account

Copilot Editor Context (read)
This application will receive bits of Editor Context (e.g. currently opened file) whenever you send it a message through Copilot Chat.

Copilot Chat (read)
This application will receive your GitHub ID, your GitHub Copilot Chat session messages (not including messages sent to another application), and timestamps of provided GitHub Copilot Chat session messages. This permission must be enabled for Copilot Extensions.

Authorizing will redirect to
https://opulent-guacamole-p74p45v796c97rq-3000.app.github.dev

Рис. 10.21. Авторизация приложения

Реализация расширения в форме наборов навыков

Агенты предоставляют мощную и гибкую архитектуру для реализации расширений Copilot, но вся эта мощь может быть избыточной. Если вам нужно, чтобы Copilot в ответ на запрос из чата вызывал конкретный инструмент или функцию API, то вы можете реализовать свое расширение в форме набора навыков.

Навыком (skill) в Copilot называется инструмент, который модель вызывает для выполнения конкретной операции в ответ на промпт, — например, для вызова внешнего API. Совокупность навыков (до пяти) называется *набором навыков*. Эти наборы позволяют интегрировать внешние сервисы или пользовательские конечные точки API в рабочий поток Copilot, не вызывая сложностей, возникающих при работе с агентами.

Наборы навыков устроены проще и требуют меньших затрат ресурсов, чем агенты. Этот вариант архитектуры лучше подходит для выполнения простых, конкретных задач, где не требуется большое количество настроек. Наборы навыков могут автоматически решать такие задачи, как маршрутизация, создание промптов, оценка функций и генерирование ответов.

На рис. 10.22 показана общая схема работы расширений в форме наборов навыков. Некоторые ее компоненты будут описаны в следующих подразделах.

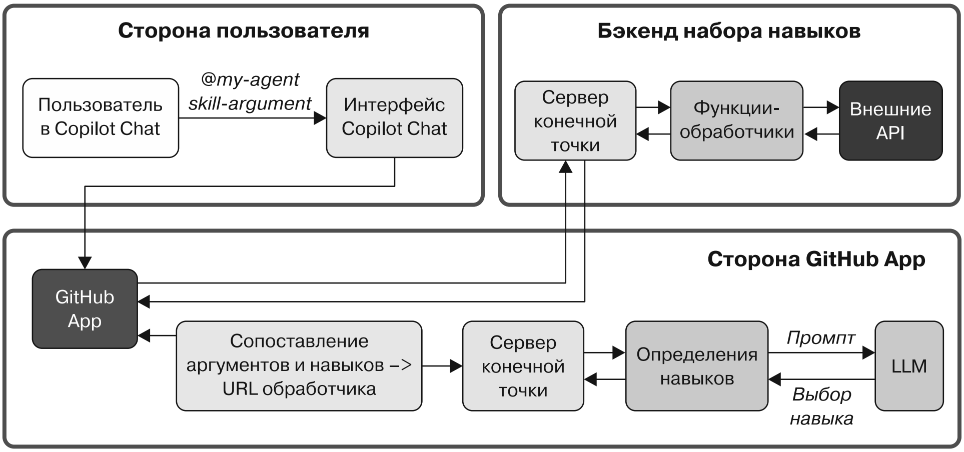


Рис. 10.22. Схема расширения в форме набора навыков

Базовая реализация

В качестве примера для этой книги я создал простое расширение Copilot, реализованное в форме набора навыков, которое позволяет выполнять три базовые операции для выпусков Go:

- определение последней версии Go;
- определение текущих поддерживаемых версий Go;
- определение того, поддерживается ли заданная версия Go или она достигла завершения своего срока жизни (EOL, End Of Life).

Код этого примера находится в репозитории книги на GitHub (https://oreil.ly/n13JY). Чтобы привести пример реализации расширения на другом языке, я написал код расширения `gover-ext` на Go. Он создан по образцу примера расширения Copilot в форме набора навыков (https://oreil.ly/WWZxW).

Файл `main.go` (https://oreil.ly/8dk9j) регистрирует функции-обработчики для конкретных URL, связанных с разными функциями расширения. Базовая логика выглядит так:

```
func main() {
    if err := run(); err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
}

func run() error {
    http.HandleFunc("/latest-version-go", handlers.LatestVersionGo)
    http.HandleFunc("/supported-versions-go", handlers.SupportedVersionsGo)
    http.HandleFunc("/is-supported-or-eol", handlers.IsSupportedOrEOL)
    http.HandleFunc("/_ping", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("OK"))
    })

    http.ListenAndServe(":8080", nil)
    return nil
}
```

Для каждого обработчика реализуется отдельная функция, которая вызывает API, проверяет ошибки и возвращает нужные результаты. Например, в обработчике для получения последней версии — LatestVersionGo (https://oreil.ly/TxXMz) — ключевые части кода реализованы так, как описаны ниже.

Сначала мы реализуем структуру для хранения версии и обращаемся с вызовом к API сайта Go, чтобы получить последнюю версию:

```
type GoVersion struct {
    Version string `json:"version"`
}

func LatestVersionGo(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Latest Go Version Called")
    req, err := http.NewRequestWithContext(r.Context(),
    http.MethodGet, "https://go.dev/dl/?mode=json", nil)
    if err != nil {
        http.Error(w, "Failed to create request",
        http.StatusInternalServerError)
        return
    }
```

После проверки ошибок, подтверждающей успешность вызовов, мы буферизуем ответ, а затем анализируем его, чтобы получить последнюю версию, и записываем ее обратно в поток:

```
// Буферизация тела ответа для декодирования
    var bodyBuffer bytes.Buffer
    tee := io.TeeReader(resp.Body, &bodyBuffer)

    // Считывание и очищение tee для последующего разбора ответа
    if _, err := io.Copy(io.Discard, tee); err != nil {
        http.Error(w, "Failed to process response body",
        http.StatusInternalServerError)
        return
    }

    // Разбор ответа JSON
    var versions []GoVersion
    if err := json.NewDecoder(&bodyBuffer).Decode(&versions);
err != nil {
        http.Error(w, "Failed to parse JSON response",
        http.StatusInternalServerError)
        fmt.Printf("Failed to decode JSON: %v\n", err)
        return
    }

    // Последняя версия хранится в первом элементе массива
    if len(versions) > 0 {
        trimmedVersion := strings.TrimPrefix(versions[0].Version, "go")
        fmt.Fprintf(w, "%s", trimmedVersion) // Версия записывается в ответ HTTP
        return
    }
```

Обработчик для поддерживаемых версий работает по похожему принципу. Имеется структура для хранения данных и вызов другой функции API для получения необработанной информации:

```
type VersionInfo struct {
    Cycle       string      `json:"cycle"`
    ReleaseDate string      `json:"releaseDate"`
    EOL         interface{} `json:"eol"`
}

func SupportedVersionsGo(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Supported Go Versions Called")
    req, err := http.NewRequestWithContext(r.Context(),
http.MethodGet, "https://endoflife.date/api/go.json", nil)
    if err != nil {
        http.Error(w, "Failed to create request",
        http.StatusInternalServerError)
        return
    }
```

Выполнив проверку ошибок, подтверждающую успешность вызова, мы буферизуем ответ, а затем анализируем результаты и преобразуем их в одну строку, которая записывается обратно в поток:

```
// Разбор JSON в срез VersionInfo
    var versions []VersionInfo
    if err := json.NewDecoder(&bodyBuffer).Decode(&versions);
err != nil {
        http.Error(w, "Failed to parse JSON response",
        http.StatusInternalServerError)
        fmt.Printf("Failed to decode JSON: %v\n", err)
        return
    }

    // Фильтрация версий с eol == false и сбор результатов
    result := make([]struct {
        Cycle       string `json:"cycle"`
        ReleaseDate string `json:"releaseDate"`
    }, 0)

    for _, v := range versions {
        // Убеждаемся, что EOL является логическим значением, равным false
        if eolBool, ok := v.EOL.(bool); ok && !eolBool {
            result = append(result, struct {
                Cycle       string `json:"cycle"`
                ReleaseDate string `json:"releaseDate"`
            }{
                Cycle:       v.Cycle,
                ReleaseDate: v.ReleaseDate,
            })
        }
    }

    // Преобразование результата в одну строку
    var builder strings.Builder
    for _, r := range result {
        builder.WriteString(fmt.Sprintf("Version: %s, Release Date:
 %s\n", r.Cycle, r.ReleaseDate))
    }

    // Теперь все версии находятся в одной строке
    if len(builder.String()) > 0 {
        fmt.Fprintf(w, "%s", builder.String()) // Запись списка поддерживаемых
                                               // версий в ответ HTTP
        return
    }
```

Схема для другой функции, определяющей, поддерживается ли версия или она достигла завершения своего срока жизни (EOL), выглядит аналогично.

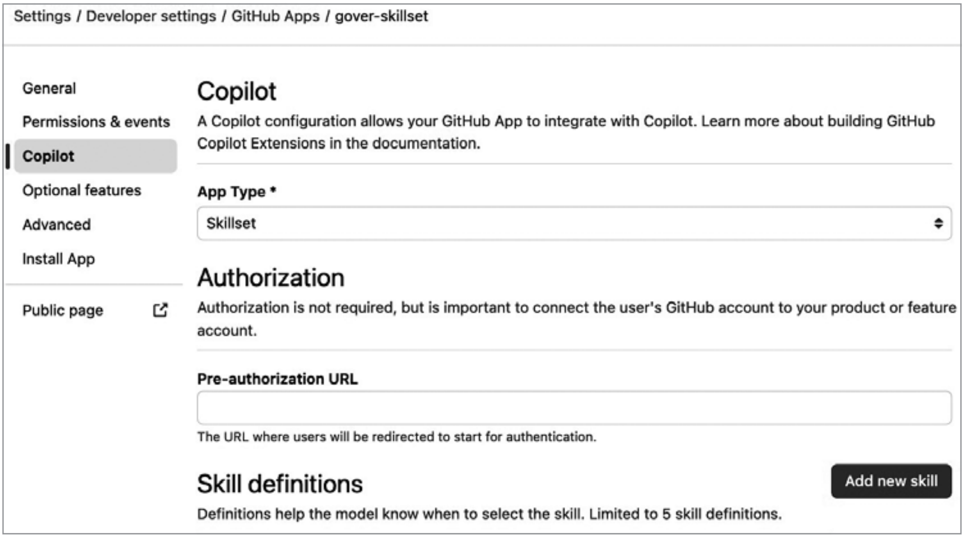
Код получается достаточно тривиальным, и в нем не упоминается ни ИИ, ни обработка промптов, поэтому возникает вопрос: как же в конечном счете промпты в Copilot Chat вызывают соответствующие обработчики? Секрет кроется в конфигурации GitHub App при создании расширения, использующего наборы навыков.

Настройка приложения GitHub App для расширения Copilot в форме набора навыков

Расширениям, реализованным в форме наборов навыков (как и расширениям, реализованным в форме агентов), необходимо приложение GitHub App, связывающее чат-интерфейс Copilot с базовой реализацией расширения. Бóльшая часть настройки выполняется так же, как и для агентов, но есть и некоторые существенные различия (создание и настройка агентов была описана в подразделе «Настройка приложения GitHub App для Copilot Agent Extension» выше в этой главе).

Первое различие заключается в том, что на странице General вам не придется задавать значение Callback URL для конкретного расширения. Вам понадобится URL обратного вызова для аутентификации, но это может простой URL вида https://github.com.

Другие, наиболее важные изменения для настройки приложения вносятся на странице Copilot. В поле App Type необходимо ввести значение Skillset. На экране появляется раздел Skill definitions (рис. 10.23).



Settings / Developer settings / GitHub Apps / gover-skillset

Copilot

A Copilot configuration allows your GitHub App to integrate with Copilot. Learn more about building GitHub Copilot Extensions in the documentation.

App Type *

Skillset

Authorization

Authorization is not required, but is important to connect the user's GitHub account to your product or feature account.

Pre-authorization URL

The URL where users will be redirected to start for authentication.

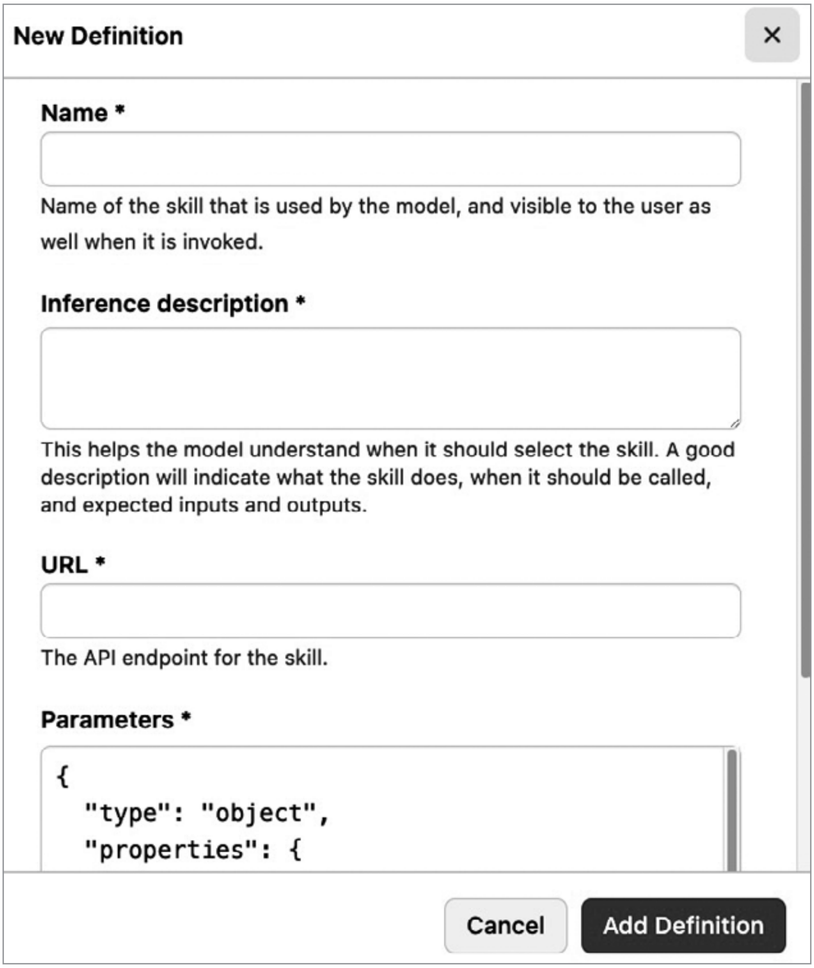
Skill definitions

Definitions help the model know when to select the skill. Limited to 5 skill definitions.

Рис. 10.23. Страница Copilot для настройки приложения набора навыков

В разделе Skill definitions настраивается сопоставление промпта в чате с обработчиками в коде. Каждый навык, который обрабатывается в коде, необходимо добавить как новый, выполнив определенную последовательность действий. Рассмотрим один пример, остальные добавляются по той же схеме.

Когда вы нажимаете кнопку Add new skill, открывается диалоговое окно с набором различных полей: Name, Inference description, URL и Parameters (рис. 10.24).



Name of the skill that is used by the model, and visible to the user as well when it is invoked.

This helps the model understand when it should select the skill. A good description will indicate what the skill does, when it should be called, and expected inputs and outputs.

The API endpoint for the skill.

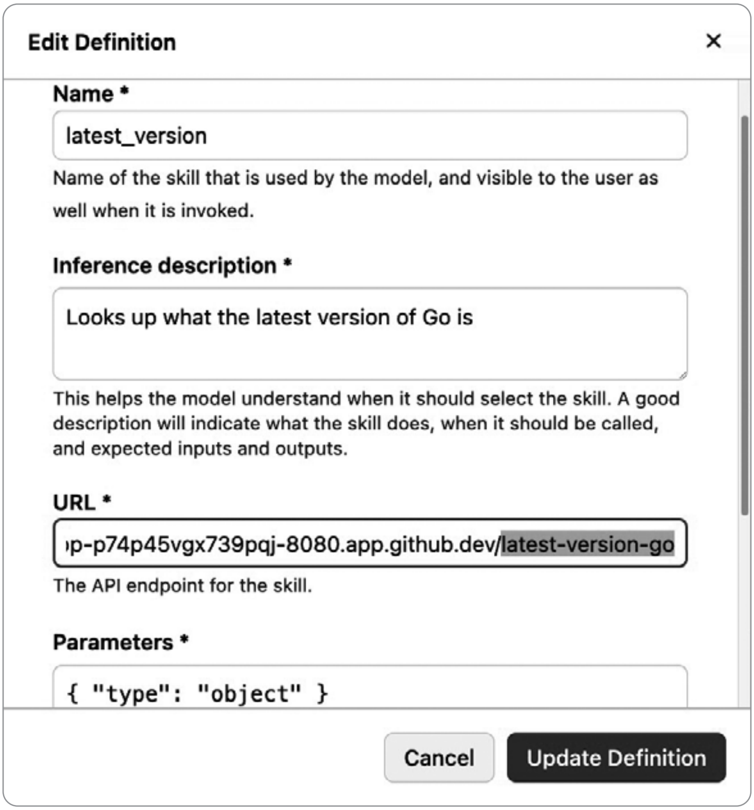
```
{
  "type": "object",
  "properties": {
```

Рис. 10.24. Новое окно определения навыков для приложения

В поле Name введите название навыка, который будет использоваться моделью и будет показываться пользователю при вызове.

В поле Inference description вы объясняете модели, какой навык надо вызвать, когда и как она должна это делать. Описание должно быть достаточно содержательным, чтобы модель могла понять, что вы имеете в виду.

Представьте, что это прямой промпт, который вы можете предоставить модели, если хотите, чтобы она выполнила этот навык. В нашем случае используется простой промпт, но в зависимости от того, что делает ваш навык и насколько он сложный, вы можете определить дополнительные параметры. Например, это могут быть ожидаемые входные и выходные данные, особенно при наличии других очень похожих наборов навыков. На рис. 10.25 показан пример определения с описанием обработки, которое соответствует одному из используемых навыков.



Name *

latest_version

Name of the skill that is used by the model, and visible to the user as well when it is invoked.

Inference description *

Looks up what the latest version of Go is

This helps the model understand when it should select the skill. A good description will indicate what the skill does, when it should be called, and expected inputs and outputs.

URL *

p-p74p45vgx739pqj-8080.app.github.dev/latest-version-go

The API endpoint for the skill.

Parameters *

```
{ "type": "object" }
```

Рис. 10.25. Определение навыка в приложении

На основании промпта в чате и того, как модель может сопоставить ее с содержимым поля Inference description, процесс вызовет конечную точку, заданную в поле URL. В этом поле вы вводите адрес сервера конечной точки с путем, указанным в коде расширения. В нашем случае, поскольку код расширения ищет `latest-version-go` для вызова обработчика, получающего новейшую версию Go, мы можем указать в этом поле конечную точку API с `latest-version-go` в конце.

В рабочей версии использовался бы специально назначенный общедоступный URL, на который будет перенаправляться запрос. Но если для этого используется ngrok, пространство Codespace или нечто похожее, то к общедоступному URL, обслуживаемому этими приложениями, нужно добавить `latest-version-go`. Например, для ngrok (https://ngrok.com) весь URL будет выглядеть так:

```
https://8fa7-2605-a601-a6ca-f00-1466-39cc-11f7-7f71.ngrok-free.app/latest-version-go
```

localhost и ngrok

ngrok позволяет создать безопасный туннель от общедоступной конечной точки в Интернете к локальному сервису на вашей машине. Благодаря этому кросс-платформенному инструменту вы сможете легко предоставлять доступ к локальным веб-серверам в Интернете для тестирования и совместного использования, не создавая сложных сетевых конфигураций.

Обычно ngrok запускается с обслуживанием на localhost:8080, но использовать localhost в конфигурации навыка не удастся, поскольку этот адрес не является общедоступным. Вместо этого необходимо получить реальный общедоступный URL, который ngrok перенаправляет на localhost:8080.

При использовании Codespace URL может выглядеть так:

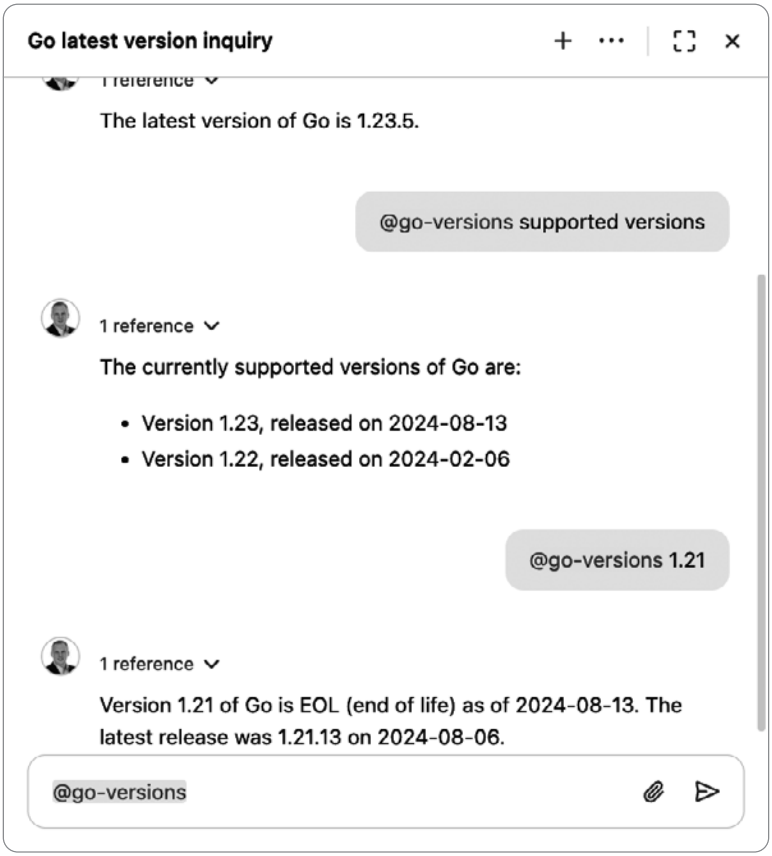
```
https://turbo-dollop-p74p45vgx739pqj-8080.app.github.dev/latest-version-go
```

В поле Parameters следует определить параметры навыка в JSON. Каждый из наших наборов навыков связан со строкой промпта, так что в это поле можно привести такое описание:

```
{ "type": "object" }
```

Когда реализация будет завершена, а приложение создано и настроено, вы можете запустить приложение так же, как ранее запускали приложение для агента. На рис. 10.26 показан пример проверки различных навыков из расширения. Обратите внимание на использование соответствующих промптов: @go-versions supported versions и @go-versions 1.21 *(справа)*.

Расширения, реализованные в форме агентов и наборов навыков, предоставляют много возможностей, поскольку могут запускаться в любом из интегрированных чат-интерфейсов (GitHub, IDE). В начале главы я упоминал еще один вариант добавления функциональности в IDE — расширения VS Code для Copilot. Его мы и рассмотрим далее.



The latest version of Go is 1.23.5.

@go-versions supported versions

The currently supported versions of Go are:

- Version 1.23, released on 2024-08-13
- Version 1.22, released on 2024-02-06

@go-versions 1.21

Version 1.21 of Go is EOL (end of life) as of 2024-08-13. The latest release was 1.21.13 on 2024-08-06.

@go-versions

Рис. 10.26. Запуск расширения go-versions

Создание расширений VS Code для Copilot

Как и расширения Copilot, расширения VS Code для Copilot предоставляют пользователю дополнительную функциональность через Copilot Chat. В отличие от расширений Copilot расширения VS Code реализуются с использованием платформы и API среды VS Code. Благодаря такому подходу к реализации расширения VS Code имеют доступ к разнообразному по возможностям и сложности функционалу VS Code и могут быть тесно интегрированы со средой. Кроме того, их использование не требует отдельного приложения GitHub App. Недостаток же заключается в том, что расширения VS Code невозможно использовать в других клиентах, не связанных с IDE (таких как чат-интерфейс в GitHub). В табл. 10.2 обновлена информация из табл. 10.1 — добавлен столбец «Расширения VS Code для Copilot».

Таблица 10.2. Сравнение типов расширений Copilot

Параметр	Агенты Copilot	Наборы навыков Copilot	Расширения VS Code для Copilot
Сложность	Требуют дополнительных усилий по управлению всем процессом взаимодействия с пользователем и генерирования ответов	Предназначены для простой интеграции с минимальными настройками	Могут быть как сложными, так и простыми в зависимости от API, используемых в VS Code
Уровень контроля	Обеспечивают полный контроль над обработкой запросов и генерированием ответов	Обеспечивают ограниченный контроль над взаимодействиями с пользователем, основное внимание направлено на получение данных и базовые действия	Обеспечивают максимальный уровень контроля, так как расширения написаны только для VS Code
Сценарии применения	Сложные сценарии, где требуется реализовать пользовательскую логику, интегрироваться с другими ИИ-моделями и управлять контекстом беседы	Простые задачи — такие как получение данных из внешнего API или выполнение базовых операций	Задачи для работы с кодом в IDE и (не обязательно) предоставление дополнительных команд для участника
Доступные интерфейсы	Все (GitHub, IDE)	Все (GitHub, IDE)	Только API среды VS Code
Поддержка	GitHub	GitHub	Команда VS Code

В конечном счете при добавлении расширения VS Code для Copilot пользователь видит нового участника чата.

Участник чата — это эксперт в предметной области, который может отвечать на вопросы по конкретной теме или выполнять определенные действия при работе с Copilot Chat в IDE. Ранее мы обсуждали эту тему в контексте участников, встроенных в Copilot (см. главу 3). Например, встроенный участник `@workspace` может отвечать в контексте всего вашего рабочего пространства VS Code. Для этого он использует интегрированные инструменты, такие как семантический поиск и индексы кода. Мы рассматривали и других участников, таких как `@terminal` и `@vscode`.

Участники чата вызываются с использованием синтаксиса `@identifier`. После того как участник будет активизирован и получит промпт, код участника может использовать один из нескольких подходов к обработке промптов. К их числу относятся:

- вызов ИИ-модели для интерпретации и ответа;
- перенаправление запроса бэкенд-сервису;
- использование встроенной логики и/или API для обработки.

Благодаря тесной интеграции с VS Code участники чата могут возвращать ответы, содержащие интерактивные элементы, такие как кнопки, деревья файлов или индикаторы выполнения. Кроме того, участники чата могут предлагать варианты дальнейших действий и слеш-команды (такие как `/explain` и `/fix` в примерах из предыдущих глав), связанные с предметной областью участника.

Итак, расширения VS Code для Copilot обеспечивают полнофункциональное взаимодействие *на стороне клиента*. Они имеют доступ к клиентскому контексту VS Code и интерфейсам фронтенда, при этом используя возможности Copilot в бэкенде.

Участники чата в других IDE

В этой главе мы говорим о VS Code, но участники чата доступны и в других IDE. Аналогично, в зависимости от возможностей базовой платформы среды, расширения VS Code для Copilot могут портироваться в другие IDE, в которых работает Copilot.

В качестве примера реализации этого расширения мы будем использовать *API Finder*. Установленное в VS Code, это расширение позволяет пользователю искать API, выполняющие определенную функцию, и передает в чат-интерфейс информацию об API и их использовании.

На рис. 10.27 показана общая схема работы расширения VS Code для Copilot. Некоторые его компоненты мы более подробно рассмотрим чуть позже.

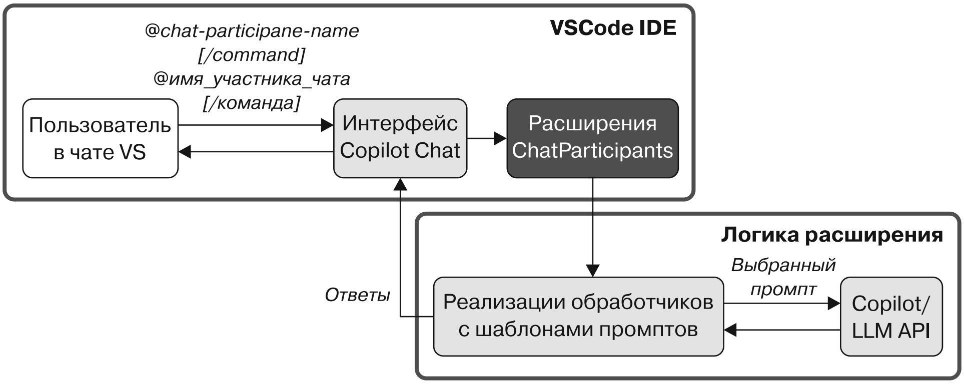
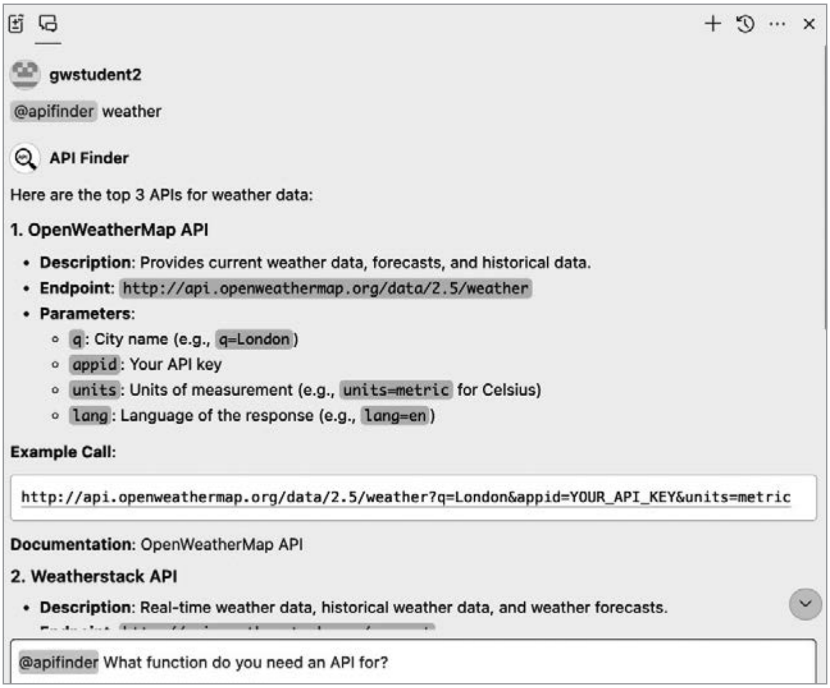


Рис. 10.27. Схема расширения VS Code

На рис. 10.28 показан пример пользовательской функциональности для поиска подходящих API.

Кроме того, расширение поддерживает дополнительную встроенную команду, которую можно вызвать как `/examples`. Пример ее использования показан на рис. 10.29.



gwstudent2

@apifinder weather

API Finder

Here are the top 3 APIs for weather data:

1. OpenWeatherMap API

- **Description**: Provides current weather data, forecasts, and historical data.
- **Endpoint**: `http://api.openweathermap.org/data/2.5/weather`
- **Parameters**:
  - `q`: City name (e.g., `q=London`)
  - `appid`: Your API key
  - `units`: Units of measurement (e.g., `units=metric` for Celsius)
  - `lang`: Language of the response (e.g., `lang=en`)

Example Call:

```
http://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY&units=metric
```

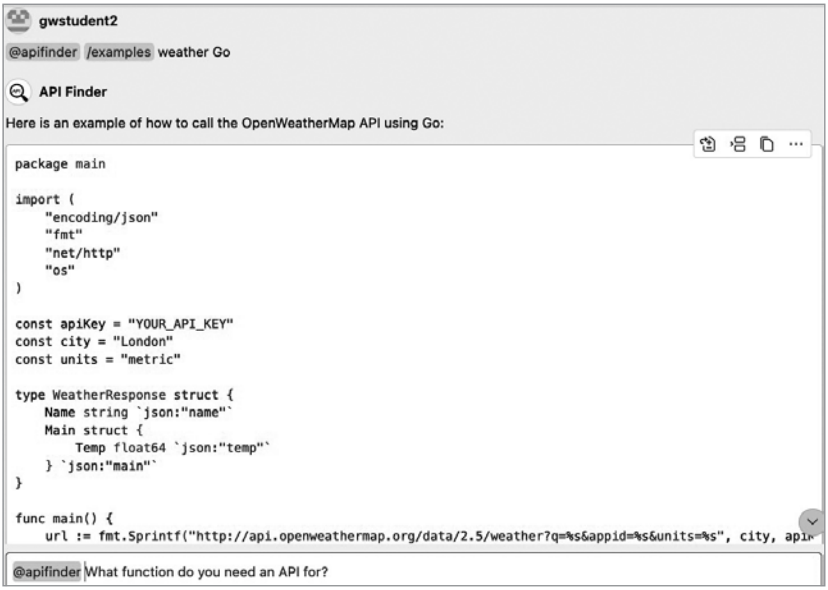
Documentation: OpenWeatherMap API

2. Weatherstack API

- **Description**: Real-time weather data, historical weather data, and weather forecasts.

@apifinder What function do you need an API for?

Рис. 10.28. Использование участника чата API Finder для поиска погодных API



gwstudent2

@apifinder /examples weather Go

API Finder

Here is an example of how to call the OpenWeatherMap API using Go:

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"
    "os"
)

const apiKey = "YOUR_API_KEY"
const city = "London"
const units = "metric"

type WeatherResponse struct {
    Name string `json:"name"`
    Main struct {
        Temp float64 `json:"temp"`
    } `json:"main"`
}

func main() {
    url := fmt.Sprintf("http://api.openweathermap.org/data/2.5/weather?q=%s&appid=%s&units=%s", city, api
```

@apifinder What function do you need an API for?

Рис. 10.29. Использование команды из участника чата

Код расширения находится в репозитории книги на GitHub (https://oreil.ly/RUAtr). Это расширение реализовано на TypeScript, главная логика находится в файле `src/extension.ts` (https://oreil.ly/VSPMM). Расширение основано на одном из примеров (https://oreil.ly/uRBxt) расширений VS Code на GitHub.

Код начинается с импортирования зависимостей из библиотек VS Code. Затем определяется промпт, который будет отправляться модели для базовой функциональности, и промпт, который будет отправляться для дополнительной команды:

```
import * as vscode from 'vscode';

const BASE_PROMPT = 'Ты используешься для поиска API. Твоя задача — найти
и предоставить документацию по трем ведущим API, соответствующим промпту.
Добавь в ответ базовую документацию об API, способах их вызова и их параметрах. Укажи
источники информации и предоставь ссылки на дополнительную информацию там, где это
уместно. Используй маркированные списки  в объяснениях там, где это возможно. Если
пользователь задаст вопрос, не относящийся к API, то вежливо откажись отвечать.';

const SAMPLE_CODE_PROMPT = 'Ты — полезный источник примеров API.
Твоя задача — найти три ведущих API, соответствующих промпту,
и предоставить простой пример кода для их вызова.
Пример должен быть коротким, насколько это возможно, но сохраняющим
практическую ценность и релевантность.
Если пользователь задаст вопрос, не относящийся к API,
то вежливо откажись отвечать.';
```

Обратите внимание: мы явно определяем, как LLM должна обрабатывать запрос, предоставляя подробные промпты.

Определение языка для /examples

В примере на рис. 10.29 мы указали язык Go во втором аргументе промпта @apifinder /examples. Но в тексте промпта язык программирования явно не указан. Это одно из преимуществ использования ИИ для обработки запросов: он вычисляет дополнительный контекст и смысл, который вам не нужно определять явно.

Код в нижней части файла создает экземпляр обработчика и добавляет значок для его использования в чате:

```
// Создание участника
const apifinder = vscode.chat.createChatParticipant("chat-tutorial.api-finder",
handler);

// Добавление значка для участника
apifinder.iconPath = vscode.Uri.joinPath(context.extensionUri, 'apifinder.jpeg');
```

Главная функция обработки выбирает промпт (в зависимости от того, требуется ли базовая функциональность, или используется команда `examples`). Затем она объединяет промпт с предыдущими сообщениями, отправляет их модели и передает ответ:

```
export function activate(context: vscode.ExtensionContext) {

    // Определение обработчика чата
    const handler: vscode.ChatRequestHandler = async (request:
    vscode.ChatRequest, context: vscode.ChatContext,
    stream: vscode.ChatResponseStream, token: vscode.CancellationToken)
     => {
        // Инициализация промпта
        let prompt = BASE_PROMPT;
        if (request.command === 'examples') {
            prompt = SAMPLE_CODE_PROMPT;
        }
        // Инициализация массива сообщений с помощью промпта
        const messages = [
            vscode.LanguageModelChatMessage.User(prompt),
        ];
        // Получение всех предыдущих сообщений участника
        const previousMessages = context.history.filter(
            (h) => h instanceof vscode.ChatResponseTurn
        );
        // Добавление предыдущих сообщений в массив сообщений
        previousMessages.forEach((m) => {
            let fullMessage = '';
            m.response.forEach((r) => {
                const mdPart = r as vscode.ChatResponseMarkdownPart;
                fullMessage += mdPart.value.value;
            });
            messages.push(vscode.LanguageModelChatMessage.Assistant(fullMessage));
        });
        // Добавление сообщения пользователя
        messages.push(vscode.LanguageModelChatMessage.User(request.prompt));
        // Отправка запроса
        const chatResponse = await request.model.sendRequest(messages, {}, token);
        // Передача ответа
        for await (const fragment of chatResponse.text) {
            stream.markdown(fragment);
        }
        return;
    };
```

Расширение тестируется достаточно просто: откройте папку в VS Code, установите зависимости, используя команду `npm install`, скомпилируйте код с помощью команды `npm run compile` и запустите расширение в новом экземпляре VS Code.

Выполнение участника в отладчике

Пожалуй, самый простой способ протестировать код участника — загрузить его в экземпляр VS Code, выполнить команды npm, а затем использовать команду Start Debugging из меню Run, чтобы запустить новый экземпляр VS Code. В новом экземпляре отладчика участник будет установлен и активен, поэтому вы можете использовать его в чат-интерфейсе экземпляра. Инструкции можно найти в файле README в репозитории (https://oreil.ly/QXsC_).

Как и в случае с другими типами расширений, перед запуском этого расширения в эксплуатацию необходимо внести некоторые изменения — например, убедиться в корректности аутентификации. В документации VS Code (https://oreil.ly/UH18y) реализация описана намного подробнее. Кроме того, в ней можно найти информацию о публикации готовых расширений в VS Code Marketplace (https://oreil.ly/HGZkA).

Резюме

Расширения GitHub Copilot и VS Code для Copilot позволяют дополнить встроенные возможности GitHub Copilot. Для доступа к этим расширениям используется интерфейс Copilot Chat и префикс `@`.

Бэкенд-код для расширений Copilot может быть двух видов. Агенты — наиболее гибкий вариант, но и самый сложный в реализации. Расширения, которые используют архитектуру набора навыков, вызывают один или несколько навыков в бэкенде, чтобы получить дополнительную информацию. Навык может быть реализован так же просто, как вызов API.

Расширения VS Code для Copilot создаются на базе платформы VS Code. Они имеют полный доступ к ее базовым API и используют ее для работы. С точки зрения пользователя они предоставляют VS Code участникам чата, похожим на встроенных участников, таких как `@workspace` и `@terminal`.

Код бэкенда для расширений может быть реализован на разных языках, в том числе JavaScript, TypeScript и Go. Для расширений, использующих форму агента и JavaScript, GitHub предоставил раннюю версию SDK, использование которой позволяет упростить код и решать задачи, характерные для рабочей версии, такие как аутентификация.

Расширениям Copilot (реализованные в форме агентов или наборов навыков) требуется приложение GitHub App для соединения чат-интерфейса Copilot и кода бэкенда. В обоих случаях используются приложения GitHub App, но их настройки различаются. Агенты используют один URL обратного вызова, заданный в приложении. В свою очередь, приложения для наборов навыков требуют дополнительного указания сведений об обработке, чтобы ИИ мог сопоставить промпт с правильной конечной точкой API навыка.

Расширения Copilot используют GitHub App, поэтому могут работать в любом чат-интерфейсе Copilot — как в IDE, так и на GitHub. Расширения VS Code для Copilot, напротив, используют API и функции VS Code, поэтому могут работать только в данной среде.

Об авторе

Брент Ластер — разработчик, руководитель, известный преподаватель и автор нескольких книг. Основатель компании Tech Skills Transformations, которая помогает людям вывести свои технические навыки на новый уровень. Автор обучающих курсов по GitHub Copilot.

Иллюстрация на обложке

На обложке изображены птицы из семейства колибри — эквадорские горные звезды *(Oreotrochilus chimborazo)*. Они обитают в основном в Эквадорских Андах, хотя встречаются и на территории Колумбии. Главная зона обитания — горные луга на высоте от 3500 до 5180 м над уровнем моря.

Эквадорская горная звезда достигает в длину до 12 см и имеет длинный изогнутый клюв. Внешний вид самцов и самок сильно различается: у самцов ярко-фиолетовый хохолок, белая грудка с черной полосой посередине и переливчатые зеленые пятна на крыльях и теле. У одного подвида на горле имеется бирюзовое пятно. Самки и молодые самцы имеют намного менее яркую внешность: зелено-коричневые хохолки и пятнистое брюшко более светлого оттенка. Эти птицы часто встречаются возле кустов чукураги, оранжевые цветы которой — один из главных источников нектара для эквадорской горной звезды.

Эти колибри считаются видом, вызывающим наименьшие опасения, так как их популяция стабильна. Многие животные, изображенные на обложках книг O’Reilly, находятся под угрозой исчезновения; все они важны для нашего мира.

Иллюстрация на обложке принадлежит Монике Камсвааг (Monica Kamsvaag) и была создана на основе старинной гравюры из книги *Royal Natural History* Ричарда Лидеккера (Richard Lydekker).

Брент Ластер

GitHub Copilot. Программирование в паре с ИИ

Перевел с английского Е. Матвеев

Научный редактор Т. Квист

Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес:
010000, Казахстан, город Астана, район Алматы, проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 04.2026. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 26.02.26. Формат 70х100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1000. Заказ 0000.