# Copilot to Agents

The complete guide to GitHub's AI developer certifications — GH-300 & GH-600

Sébastien Brochet

August 2026

# Contents

# Disclaimer

This book is an independent study guide. It is **not** affiliated with, endorsed by, or sponsored by GitHub, Inc. or Microsoft Corporation. “GitHub”, “GitHub Copilot”, “Microsoft”, and related names and logos are trademarks of their respective owners and are used here for identification and educational purposes only.

## Exam objectives evolve

Certification objectives change over time. This book targets:

- **GH-300 — GitHub Copilot**: the skills measured **as of August 7, 2026**.
- **GH-600 — Developing in Agentic AI Systems**: the current agentic-SDLC skill set at the time of writing.

Microsoft and GitHub update exams periodically — typically to reflect new features and to retire deprecated ones. **Always confirm the current objectives** on the official study guides before you sit either exam:

> **Source**: Study guide for Exam GH-300: GitHub Copilot
>
> **Source**: Study guide for Exam GH-600: Developing in Agentic AI Systems

## Preview vs general availability

GitHub Copilot and its agentic capabilities move quickly between **Preview** and **general availability (GA)**. A feature described here as Preview may be GA (or renamed, or replaced) by the time you read it — and the reverse can happen too. The exams note that most questions cover GA features, with occasional questions on commonly used Preview features. Where a feature’s status matters, this book flags it and tells you to verify.

## Commands and configuration

This is a hands-on book: it shows real commands (gh, GitHub Copilot CLI), configuration files (instructions files, prompt files, MCP configuration), and workflow snippets. Product surfaces, flags, and file paths change. **Verify any command or configuration against the current GitHub documentation** before depending on it in your own environment.

## Two Copilots

Throughout, “Copilot” means **GitHub Copilot** — the developer tool. It is distinct from **Microsoft 365 Copilot**, the business productivity assistant covered in the sibling book *Copilot to Transformation*. The exams in this book concern GitHub Copilot; do not conflate the two.

## No warranty

The information here is provided “as is”, without warranty of any kind. Passing an exam depends on many factors, including the current exam form and your own preparation. Use this book as one part of a broader study plan that includes hands-on practice and the official Microsoft Learn and GitHub resources.

## License

# Introduction

Artificial intelligence has moved from autocomplete to autonomy. In a few short years, GitHub Copilot went from suggesting the next line of code to reviewing pull requests, running in your terminal, and — most recently — operating as an **agent** that plans work, uses tools, and opens pull requests on its own. GitHub, through Microsoft's certification program, now offers two credentials for the developers and engineers at the center of this shift. This book prepares you for both:

- **GH-300 — GitHub Copilot**: for the developer who wants to *use* Copilot fluently and responsibly across the IDE, the CLI, chat, and Agent Mode.
- **GH-600 — Developing in Agentic AI Systems**: for the engineer who *builds, supervises, and governs* autonomous agents inside the software development lifecycle (SDLC), using GitHub as the system of record and control plane.

Unlike the sibling volume *Copilot to Transformation* (for the no-code business certifications AB-730 and AB-731), this book is **hands-on**. It assumes you write code and know your way around Git and GitHub, and it shows real commands and configuration — Copilot CLI, MCP servers, instructions and prompt files, custom agents, and CI workflows — alongside the concepts the exams test.

## Who this book is for

You already ship software, and you want to use GitHub Copilot as well as it can be used — or you are moving from *using* AI to *engineering* it: wiring up tools, giving agents memory, evaluating their output, and putting guardrails around what they are allowed to do. If either describes you, this book is written for you. A development background is assumed; no prior agent-building experience is required.

## How the two exams relate

The two certifications share a foundation and then diverge:

Rather than repeat the common ground twice, this book teaches it **once** in Part I, then gives each exam its own dedicated track. Every exam objective is mapped to a chapter — see Annex C for the full traceability.

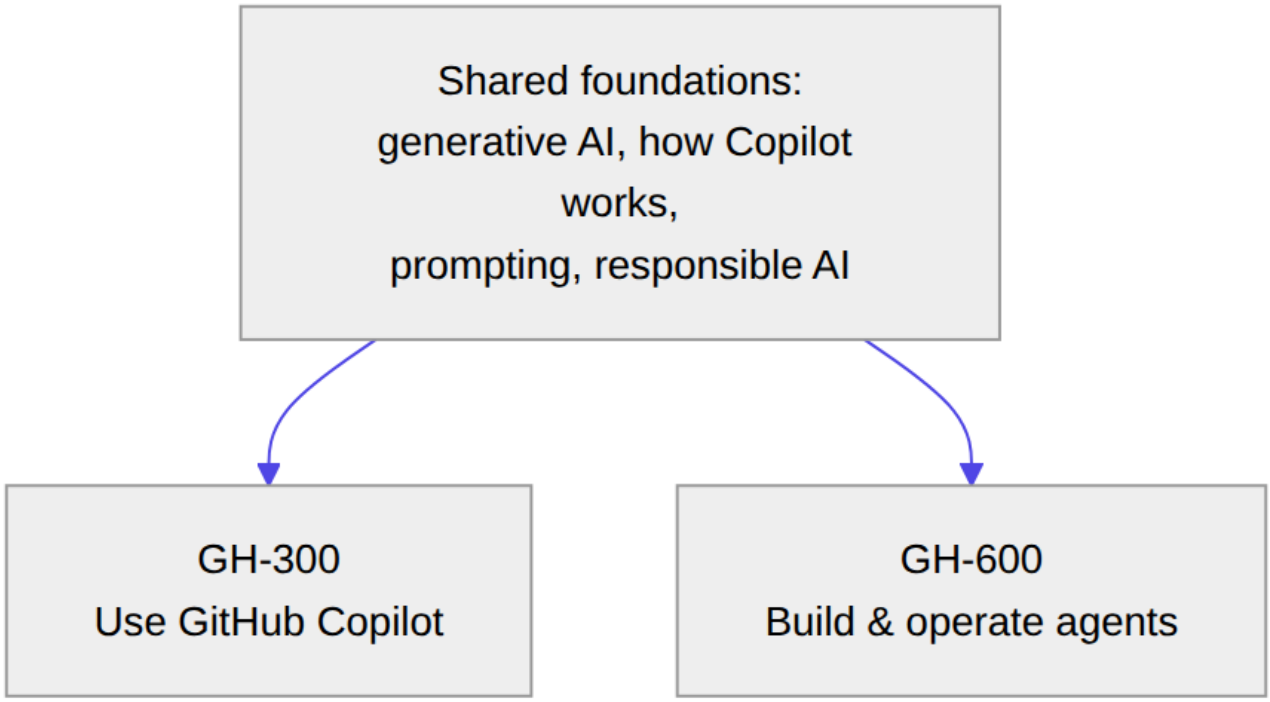


Figure 1: Diagram

## How this book is organized

- **Part I — AI foundations for developers** (shared): what generative AI is, how GitHub Copilot works, prompt engineering, and responsible AI.
- **Part II — GH-300 track**: Copilot in the IDE and CLI; Agent Mode, Copilot Edits, MCP, and code review; developer productivity; and administering Copilot (policies, privacy, safeguards).
- **Part III — GH-600 track**: agent architecture and SDLC integration; tools and MCP; memory, state, and execution; evaluation and tuning; multi-agent orchestration; and guardrails and accountability.
- **Part IV — Exam readiness**: objective checklists, high-yield facts, and a full mock exam for each certification.

## How to use this book

Each chapter opens with **"In 30 seconds"** and an **exam map** telling you exactly which objectives it covers. Along the way you will find callouts: key concepts, how it works, exam tips, pitfalls, tips, definitions, hands-on commands, and sources that link back to the official documentation. Every chapter ends with **practice questions**; each track ends with a **mock exam** of roughly 40–50 questions.

> **Tip**: read for understanding first, then try the hands-on snippets in a scratch repository. If you can explain a capability in your own words, run it, *and* pick the right answer, you know it.
>
> **Prefer to read offline?** Download the latest **PDF or EPUB** — rebuilt automati-

cally from the same sources on every update.

## A 4-week study plan

A steady, four-week rhythm works well for most readers. Adjust to your pace.

| Week | Focus | Chapters | Goal |
|---|---|---|---|
| **1** | Foundations (both exams) | Part I (1–4) | Understand generative AI, how Copilot works, prompting, and responsible AI. Do all practice questions. |
| **2** | GH-300 track | Part II (5–8) | Master Copilot in the IDE/CLI, its advanced capabilities, productivity workflows, and administration. |
| **3** | GH-600 track | Part III (9–14) | Master agent architecture, tools/MCP, memory, evaluation, multi-agent orchestration, and guardrails. |
| **4** | Exam readiness | Part IV (15–16) + annexes | Take both mock exams, review weak areas, skim the glossary and product reference. |

> **Exam tip**: a passing score is **700** on each exam. GH-300 targets the skills measured **as of August 7, 2026**. Before you book, open the official study guide, confirm the objectives haven't changed, and check which features are now generally available.

## A note on accuracy

AI products evolve quickly, and features move between Preview and general availability — especially in the fast-moving world of agents. This book is grounded in official Microsoft Learn and GitHub documentation and flags where things are likely to change. Commands and configuration are written against current GitHub Docs, but always confirm against the live documentation before relying on a specific flag or key. When in doubt, the study guide and product docs are the final word — and the sources throughout point you there.

Let's begin.

# Chapter 1 — Understanding Generative AI for Developers

*Part I — AI foundations for developers*

---

## In 30 seconds

- **The core idea**: large language models (LLMs) predict the next token from patterns learned in training; GitHub Copilot applies this to *code* with the context of your editor and repository.
- **Why it matters**: understanding what an LLM can and cannot do is the foundation for using Copilot well and for reasoning about agents later.
- **The exam angle**: GH-300 tests the **limitations of LLMs and Copilot**; both exams assume you grasp tokens, context, and probabilistic generation.
- **Remember**: Copilot *predicts plausible code*, it does not *know* correct code — you stay the author.

---

## Exam map

**Exam map — GH-300 · Understand GitHub Copilot data and architecture (LLM limitations) · Foundation for GH-600**

---

## 1. Key concepts

GitHub Copilot feels like magic the first time an entire function appears from a comment. But there is no magic — there is a **large language model** doing one thing very well: predicting the next piece of text. Understanding that single mechanism, and its consequences, is the foundation for everything else in this book. It is also directly tested: GH-300's data-and-architecture area asks you to describe the **limitations of LLMs and Copilot**, and the whole GH-600 track assumes you know what a model can and cannot do on its own.

> **Definition — Large language model (LLM)**: a neural network trained on vast amounts of text and code to predict the most probable next **token** given the to-

kens so far. Modern LLMs use the *transformer* architecture, which lets the model weigh how much each earlier token should influence the next one.

**Definition — Token**: the unit an LLM reads and writes — roughly a word fragment. `getUserById` might be several tokens (`get`, `User`, `By`, `Id`). Tokens matter because models have a finite **context window** (a maximum number of tokens they can consider at once) and because usage is often measured and billed in tokens.

## Generative AI, in one sentence

**Generative AI** produces new content — text, code, images — rather than merely classifying or retrieving existing data. When you ask Copilot to write a function, it is not searching a database of functions and copying one out; it is **generating** a sequence of tokens that, statistically, looks like correct code for your context.

**Key concept**: Copilot predicts *plausible* code, not *verified* code. A suggestion is the model's best guess at what a competent developer would write next — which is often excellent, sometimes subtly wrong, and occasionally invented outright. You remain the author and the reviewer.

## Pretraining, then specialization

An LLM is **pretrained** once, at great expense, on a large corpus. That frozen set of learned weights is then used for **inference** — the fast, cheap step that happens every time you type. Two consequences follow immediately and both are testable:

- **Knowledge cutoff**: the base model only "knows" what existed in its training data. A brand-new library released last week is invisible to the model unless that information is supplied at inference time through context (open files, documentation you reference, or tools).
- **No live understanding**: the model has no awareness of your intent, your runtime, or whether its output compiles. It pattern-matches; it does not execute.

**Definition — Context window**: the maximum number of tokens (prompt + response) a model can process in a single request. Everything the model "sees" about your problem must fit inside it — which is why *what context Copilot gathers* (Chapter 2) and *how you craft prompts* (Chapter 3) matter so much.

## The two Copilots — do not conflate them

**Pitfall**: **GitHub Copilot** (the developer tool this book covers — IDE, CLI, agents, MCP) is a different product from **Microsoft 365 Copilot** (the business assistant in Word, Teams, and Outlook). They share the underlying idea of grounding an LLM in your context, but their surfaces, data handling, and exams are distinct. The GH-300 and GH-600 exams concern **GitHub Copilot**.

---

## 2. How it works

### Next-token prediction, one step at a time

Given a sequence of tokens, the model outputs a probability distribution over every possible next token, picks one, appends it, and repeats. A `for` loop, a variable name, a closing brace — each is chosen because it is the most probable continuation of everything before it, including the code already in your file.

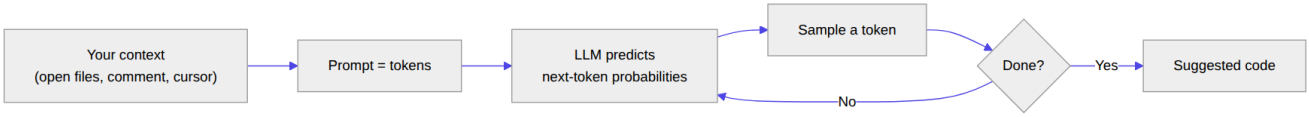


Figure 2: Diagram

> **How it works**: because the model samples from a probability distribution, generation is **non-deterministic** by default. A setting often called *temperature* controls how adventurous the sampling is: low temperature favors the single most likely token (predictable, repetitive), higher temperature allows less likely tokens (more varied, more creative — and more error-prone). This is why the *same* prompt can yield *different* suggestions on two tries.

### Why context is everything

The model's only knowledge of *your* problem is the tokens you give it. Copilot's real skill is assembling good context — the current file, related open tabs, the file's path and language — into a prompt before the model ever runs. Chapter 2 traces that assembly in detail. The practical takeaway for now:

> **Tip**: keep relevant files open and give clear, local cues (a descriptive function name, a leading comment, a type signature). You are not "telling the model what to do" so much as *shaping the probabilities* toward the code you want.

### Capabilities and limitations, side by side

| The model is strong at… | The model is unreliable at… |
|---|---|
| Boilerplate, idiomatic patterns, common algorithms | Facts it was never trained on (new APIs, private code) |
| Translating between languages and styles | Guaranteeing code compiles, runs, or is secure |
| Explaining and summarizing existing code | Arithmetic, precise counting, exact version numbers |
| Drafting tests, docs, and sample data | Knowing *your* business rules or runtime state |

> **Definition — Hallucination (fabrication)**: a confident, plausible-sounding output that is factually wrong — an invented function, a non-existent flag, a misremembered API. Hallucinations are not bugs to be fully "fixed"; they are an inherent property of probabilistic generation, which is exactly why validation (Chapter 4) is mandatory.

---

## 3. In the real world

**Scenario — a helpful suggestion and a hidden trap.** A developer writes a comment, `// parse an ISO-8601 date and return the Unix timestamp`, and Copilot instantly produces a clean function. It compiles and passes the happy-path test. But the suggestion silently assumes the local time zone, and for dates without an explicit offset it returns a value that is wrong by hours. The model produced *plausible* code — the shape of a correct answer — without *knowing* the correct answer.

The developer who understands next-token prediction is not surprised. She treats the suggestion as a first draft: she reads it, adds a test for a UTC-vs-local edge case, and fixes the assumption. The one who thinks Copilot "knows" how to parse dates ships the bug. Same tool, different mental model — and that mental model is exactly what the exam probes.

---

## 4. Exam tips

> **Exam tip**: when a question asks about the **limitations of LLMs/Copilot**, the defensible answers cluster around: it can **fabricate**, it has a **knowledge cutoff**, it is **non-deterministic**, it does **not execute or verify** code, and it is bounded by a **context window**. Answers implying Copilot "understands" intent or "guarantees" correctness are traps.
>
> **Exam tip**: know the vocabulary — **token**, **context window**, **pretraining vs inference**, **hallucination/fabrication**, **non-deterministic output**. GH-300 uses these terms directly.
>
> **Exam tip**: any answer that conflates **GitHub Copilot** with **Microsoft 365 Copilot** is wrong on these exams. Keep them separate.

---

## 5. Common pitfalls

> **Pitfall**: treating a suggestion as verified truth. Plausible ≠ correct. The model optimizes for "looks right," not "is right."

- **"It knows my codebase"**: it only knows the context supplied at inference time. Nothing more is visible to the model.
- **"Same prompt, same answer"**: generation is probabilistic; expect variation, and don't rely on a specific suggestion reappearing.
- **"Newer is in there"**: the base model can't know about libraries or events after its training cutoff unless you supply that information as context.
- **"More context is always better"**: context is finite (the window) and noisy context can crowd out the signal. Relevant beats voluminous.

---

## 6. Practice questions

**1.** Which statement best describes how an LLM like the one behind GitHub Copilot generates code?

- A. It searches a database of code snippets and returns the closest match.
- B. It predicts the most probable next token given the preceding context, one token at a time.
- C. It compiles candidate programs and returns the one that passes your tests.
- D. It queries the internet in real time for the latest documentation.

Answer

**Correct: B.** LLMs perform next-token prediction from learned probabilities. A describes retrieval, not generation; C implies execution/verification the model does not perform; D describes live web access, which the base model does not have.

**2.** A developer runs the same Copilot prompt twice and gets two different functions. Why?

- A. Copilot is broken and should be reinstalled.
- B. Generation is probabilistic; the model samples from a distribution, so output can vary.
- C. The developer's license expired between attempts.
- D. The context window doubled on the second run.

Answer

**Correct: B.** Sampling from a probability distribution makes output non-deterministic by default. A, C, and D are unrelated to why suggestions vary.

**3.** Which of the following is a genuine limitation of GitHub Copilot's underlying model?

- A. It can fabricate plausible but incorrect code (hallucination).
- B. It always refuses to generate code it hasn't seen before.
- C. It guarantees that generated code compiles.
- D. It has real-time awareness of your program's runtime state.

Answer

**Correct: A.** Hallucination is inherent to probabilistic generation. B is false (the model generates novel combinations constantly); C and D describe capabilities the model does not have.

**4.** Why does keeping relevant files open in your editor tend to improve Copilot's suggestions?

- A. It increases the model's temperature setting.
- B. It gives Copilot more relevant context to include in the prompt, shaping predictions toward your code.
- C. It retrains the model on your repository.
- D. It disables the content filters.

Answer

**Correct: B.** Copilot builds the prompt from available context; relevant open files supply useful tokens. It does not change temperature (A), retrain the model (B is about inference-time context, not training — C is wrong), or affect filtering (D).

**5.** A question refers to “Copilot” summarizing a Word document in Teams. Within the scope of GH-300 and GH-600, why is this out of scope?

- A. GitHub Copilot cannot summarize anything.
- B. That describes Microsoft 365 Copilot, a different product from GitHub Copilot.
- C. Summarization is a Preview-only feature.
- D. Word documents exceed every context window.

Answer

**Correct: B.** Word/Teams summarization is Microsoft 365 Copilot territory; these exams concern GitHub Copilot. A is false, and C/D are fabricated rationales.

---

## Further reading

- **Chapter 2 — How GitHub Copilot Works**: how context becomes a prompt, and the filtering the request passes through.
- **Chapter 4 — Using AI Responsibly**: why non-determinism and hallucination make validation non-optional.

  **Source**: Study guide for Exam GH-300: GitHub Copilot

  **Source**: GitHub Copilot Fundamentals — Part 1 (Microsoft Learn)

# Chapter 2 — How GitHub Copilot Works

*Part I — AI foundations for developers*

---

## In 30 seconds

- **The core idea**: Copilot builds a prompt from your context, sends it through a filtering proxy to an LLM, and post-processes the result before showing a suggestion.
- **Why it matters**: knowing the data flow explains privacy behavior, latency, and why suggestions vary.
- **The exam angle**: GH-300's data-and-architecture area tests **data usage/flow/sharing**, **input processing and prompt building**, **proxy filtering and post-processing**, and the **suggestion lifecycle**.
- **Remember**: prompt building → proxy filtering → model → post-processing → suggestion.

---

## Exam map

**Exam map — GH-300 · Understand GitHub Copilot data and architecture (data handling & flow; lifecycle)**

---

## 1. Key concepts

Chapter 1 established what a language model does. This chapter follows a single keystroke through the machinery that turns your editor context into a suggestion. GH-300 dedicates a whole skill area to it — **Understand GitHub Copilot data and architecture** — and the questions are concrete: what data is sent, how the prompt is built, what the proxy filters, and the order of the suggestion lifecycle.

> **Definition — Prompt (in Copilot's sense)**: the bundle of context Copilot assembles and sends to the model. It is more than what you typed: it can include the

surrounding code, the file's language and path, and snippets from related open tabs. You rarely write the prompt directly — Copilot constructs it for you.

**Definition — Proxy**: a GitHub-operated service that sits between your editor and the model. Requests and responses pass through it so that filtering (content safety, optional public-code matching) and other processing can be applied consistently.

### Three data flows, kept separate

Copilot handles three kinds of data, and the exam expects you to tell them apart:

- **Prompts** — the context sent to generate a suggestion.
- **Suggestions** — what the model returns.
- **Engagement/usage data** — telemetry about acceptances, so features can be measured and improved.

**Key concept**: for **Copilot Business** and **Copilot Enterprise**, prompts and suggestions are used to produce the response and are **not retained** to train the foundation models, and your code is not used as training data. Retention and settings can differ for individual plans, and administrators can further restrict data with content exclusions (Chapter 8). Always confirm current terms in the GitHub Trust Center before making a compliance claim.

---

## 2. How it works

### The journey of one suggestion

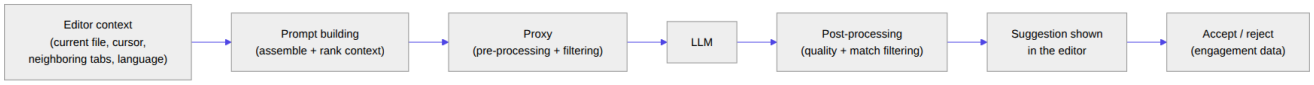


Figure 3: Diagram

**1. Input processing and prompt building.** Copilot gathers signals from your session — the code before and after the cursor, the file path and language, and relevant snippets from other open tabs ("neighboring tabs") — and assembles them into a prompt that fits the model's context window. Better local context (clear names, a leading comment, an open related file) produces a better prompt, which is the whole basis of prompt crafting in Chapter 3.

**How it works**: Copilot ranks and trims context to fit the window. It cannot send your entire repository; it sends a curated, bounded slice. That is why *which files are open* measurably changes suggestions.

**2. Proxy filtering.** The request passes through GitHub's proxy, which applies safety filtering (for toxic or harmful content) and, on the way back, can apply the optional **duplication detection filter** that suppresses suggestions matching public code (Chapter 8). The proxy is also where organizational content exclusions are enforced so that excluded files never reach the model.

**3. Model inference.** The LLM performs next-token prediction (Chapter 1) and returns candidate text.

**4. Post-processing.** Copilot filters low-quality or unsafe candidates, applies the public-code match filter if enabled, and formats what remains before showing it inline or in chat.

**5. You decide.** Accepting or rejecting produces engagement data. The code you accept is yours to review, test, and own (Chapter 4).

> **Definition — Code-suggestion lifecycle**: the ordered path *context → prompt building → proxy filtering → model → post-processing → suggestion → accept/reject*. Knowing this order — especially that filtering happens **around** the model, not inside it — is a common exam target.

### Where privacy behavior comes from

Because the proxy and prompt-building stages are explicit, the privacy story is explainable rather than magical: content exclusions stop files from ever being sent; the duplication filter stops certain outputs from being shown; Business/Enterprise terms govern retention. Each control maps to a specific stage of the flow.

> **Tip**: when reasoning about "can Copilot see file X?", think in terms of the flow — is X in the gathered context, and is it excluded at the proxy? That framing answers most privacy questions.

---

## 3. In the real world

**Scenario — debugging a "why did it suggest that?" moment.** A developer notices Copilot proposing a helper that clearly came from *another* file in the project. A teammate assumes Copilot "indexed the whole repo." The developer who knows the flow explains it correctly: that other file was **open in a neighboring tab**, so its content entered the gathered context and shaped the prompt. Close the tab, and the suggestion changes. Later, the team enables content exclusions for a folder of secrets; those files now never reach the proxy, so Copilot can no longer draw on them. No magic — just the pipeline, understood.

---

## 4. Exam tips

> **Exam tip**: memorize the **lifecycle order** and that **filtering happens at the proxy and in post-processing — around the model**, not inside it. Questions like "where is the public-code match filter applied?" hinge on this.

> **Exam tip**: distinguish the three data types — **prompts**, **suggestions**, **engagement data** — and know that under **Business/Enterprise**, code and prompts are **not used to train** the models.

> **Exam tip**: "neighboring tabs" is real. If asked why an unrelated-looking suggestion appeared, *related open files entering the context* is usually the intended answer.

---

## 5. Common pitfalls

**Pitfall**: believing Copilot "reads your whole repository." It sends a **bounded, ranked slice** of context that fits the window — not the entire codebase.

- **Confusing exclusions with the duplication filter**: **content exclusions** limit what Copilot can *see* (input side); the **public-code match filter** limits what it can *show* (output side). Different stages, different purposes.
- **Assuming code is always used for training**: under Business/Enterprise it is not. Don't over- or under-state retention — cite the Trust Center.
- **Thinking filtering is "inside" the model**: it is applied by the proxy and post-processing.

---

## 6. Practice questions

**1.** In what order does a Copilot code suggestion flow?

- A. Model → prompt building → proxy → suggestion
- B. Context gathering → prompt building → proxy filtering → model → post-processing → suggestion
- C. Proxy → model → context gathering → suggestion
- D. Prompt building → model → context gathering → proxy

Answer

**Correct: B.** Context is gathered and built into a prompt, filtered at the proxy, run through the model, post-processed, then shown. The other orderings misplace one or more stages.

**2.** A suggestion appears that resembles code from a file the developer has open but isn't editing. The most likely explanation is:

- A. Copilot trained on the repository overnight.
- B. The open file was included as neighboring-tab context in the prompt.
- C. The proxy injected the code.
- D. The duplication filter added it.

Answer

**Correct: B.** Related open tabs contribute to the gathered context. A misdescribes training; C and D misattribute a filtering stage as a source of content.

**3.** Which control prevents specific files from ever being sent to the model?

- A. The public-code duplication filter
- B. Content exclusions
- C. Temperature settings
- D. The knowledge cutoff

Answer

**Correct: B.** Content exclusions act on the **input** side, keeping files out of the context. The duplication filter (A) acts on outputs; C and D are unrelated.

**4.** Under Copilot Business/Enterprise, how are your prompts and code used?

- A. They are retained to train the foundation models.
- B. They are used to generate the response and are not used to train the foundation models.
- C. They are published to public repositories.
- D. They are shared with other customers for benchmarking.

Answer

**Correct: B.** Business/Enterprise terms state prompts and code are used to produce the response and not to train the models. A, C, and D contradict those terms.

**5.** Where is the filter that suppresses suggestions matching public code applied?

- A. Inside the model's weights
- B. In the proxy / post-processing, around the model
- C. In the editor's syntax highlighter
- D. During pretraining

Answer

**Correct: B.** Match filtering is applied around the model (proxy/post-processing), not within it (A), not by editor tooling (C), and not at training time (D).

---

## Further reading

- **Chapter 1 — Understanding Generative AI for Developers**: the model at the center of this pipeline.
- **Chapter 8 — Administering Copilot**: content exclusions and the public-code duplication filter in depth.

  **Source**: How GitHub Copilot handles your data (GitHub Trust Center)

  **Source**: GitHub Copilot plans and features

# Chapter 3 — Prompt Engineering & Context Crafting

*Part I — AI foundations for developers*

---

## In 30 seconds

- **The core idea**: good prompts give Copilot clear intent and the right context; how you structure a request materially changes the output.
- **Why it matters**: prompt and context crafting is the highest-leverage skill for day-to-day Copilot use.
- **The exam angle**: GH-300 tests **prompt structure and context**, **how context is determined**, **zero-shot and few-shot prompting**, **best practices**, and **prompt process flow / chat-history usage**.
- **Remember**: be specific, provide context, show examples (few-shot), iterate.

---

## Exam map

**Exam map — GH-300 · Apply prompt engineering and context crafting**

---

## 1. Key concepts

Chapter 2 showed that Copilot builds a prompt from context before the model runs. Prompt engineering is the skill of **steering that context and your instructions** so the model's probabilities bend toward the code you actually want. GH-300 tests it directly: prompt structure and context, how context is determined, zero-shot versus few-shot prompting, best practices, and how the chat conversation carries history.

> **Definition — Prompt engineering**: the practice of designing and refining the instructions and context you give an AI — what you ask, how you phrase it, and what you reference — to get more accurate, relevant, and useful output.
>
> **Definition — Context**: everything the model "sees" for your request — your explicit words plus the code Copilot gathers (current file, selection, neighboring

tabs) and anything you deliberately reference. Weak output is very often a *context* problem, not a *model* problem.

### Anatomy of a strong prompt

A good developer prompt tends to carry four things. You will not always spell out all four, but the higher-stakes the task, the more each one matters:

- **Intent (goal)** — what you want produced ("write a function that…", "refactor this to…").
- **Context** — the relevant code, constraints, and background ("this runs in a browser, no Node APIs").
- **Specifics** — inputs, outputs, edge cases, libraries, and style ("return a `Result`, handle empty input").
- **Examples** — a sample input/output or a pattern to imitate.

**Key concept**: be specific and give the model something to imitate. "Make this better" gives the model nothing to optimize for. "Extract the validation into a pure function, add JSDoc, and keep the public signature unchanged" gives it a target.

### Zero-shot vs few-shot

**Definition — Zero-shot prompting**: asking for the result with **no example** — you rely on the model's general training ("Write a regex for a UK postcode").

**Definition — Few-shot prompting**: including **one or more examples** of the desired input/output or style so the model imitates the pattern ("Given these three sample rows, generate ten more in the same shape"). Few-shot is the go-to when format or style must be exact.

---

## 2. How it works

### How Copilot determines context

Copilot assembles context from several sources, and you can influence each:

- **Implicit context** — the current file around your cursor, your active selection, and related open tabs (Chapter 2). Managing which files are open is itself a form of prompting.
- **Explicit references** — in Copilot Chat you can point at specific context: chat variables such as `#file`, `#selection`, or the whole workspace; participants such as `@workspace`; and `/` slash commands such as `/explain`, `/fix`, `/tests`. (Exact syntax varies by IDE — verify in your editor's docs.)
- **Conversation history** — in a chat thread, earlier turns remain in context, so follow-ups like "now add error handling" build on what came before.

**How it works**: prompting is a **loop**, not a single shot. Because generation is probabilistic (Chapter 1), the reliable path is: draft a focused prompt → read the result → refine (add a constraint, an example, or a reference) → repeat. The conversation carries context, so refining usually beats starting over.

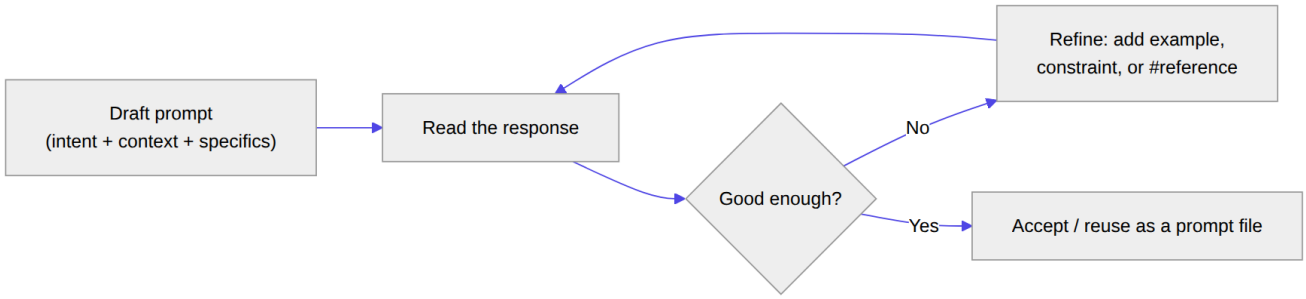


Figure 4: Diagram

### Best practices that reliably help

- **One job per prompt** — ask for a single, well-scoped thing; chain follow-ups rather than cramming.
- **Give clear intent and specifics** — name inputs, outputs, edge cases, and the library or style to use.
- **Provide examples (few-shot)** when the format matters — sample data, a target signature, a style to copy.
- **Keep relevant files open** — you are supplying context (Chapter 2).
- **Iterate** — treat the first answer as a draft; "make it shorter," "handle nulls," "add a test."
- **Follow good coding practice in the prompt** — clear names and comments in your code steer better output.

  **Hands-on**: in Copilot Chat, scope the request explicitly and ask for one thing:

  ```
  /tests #selection
  Write unit tests for the selected function. Cover empty input, a single element,
  and a duplicate-key case. Use the project's existing test framework.
  ```

---

## 3. In the real world

**Scenario — from vague to precise.** A developer types `// sort the list` and gets a generic sort that ignores the domain. Reframed with intent, specifics, and an example, the prompt becomes:

> "Sort `orders` by `priority` descending, then by `createdAt` ascending. `priority` is one of `low|medium|high`. Return a new array; don't mutate the input. Example: a `high` order from yesterday should come before a `high` order from today."

Copilot now produces a comparator with the right tie-breaker and an explicit priority ordering — because the prompt supplied the intent, the constraints, and a concrete example to imitate. In a follow-up turn, "now add a unit test for the tie-break case" reuses the conversation's context instead of re-explaining.

---

## 4. Exam tips

**Exam tip**: know the difference between **zero-shot** (no example) and **few-shot** (one or more examples) prompting — and that few-shot is preferred when the output's **format or style** must match.

**Exam tip**: when a weak prompt yields a weak answer, the best fix is usually to **add context or an example**, or to **reference the relevant file/selection** — not to repeat the same prompt or switch models.

**Exam tip**: iteration and **conversation history** are features. Refining within the same chat thread is often the intended "best next step" because earlier turns remain in context.

---

## 5. Common pitfalls

**Pitfall**: the "one perfect prompt" myth. Strong results come from specificity and iteration, not from a single magic sentence.

- **Vagueness**: "make it better" — better *how*? Give a target.
- **No references**: asking about "the config" without referencing the file forces a generic guess.
- **Overstuffing**: a rambling, multi-goal prompt buries the intent and wastes the context window.
- **Ignoring history**: starting a brand-new chat for a follow-up throws away useful context.
- **Wrong context open**: unrelated open tabs can pull suggestions off-target — relevance beats volume.

---

## 6. Practice questions

**1.** Which prompt is most likely to produce correct, well-shaped code?

- A. "Fix this."
- B. "Improve the function."
- C. "Refactor `parseConfig` to return a `Result<Config, Error>`, keep the signature, and add a test for a malformed file."
- D. "Make it faster somehow."

Answer

**Correct: C.** It states intent, specifics, and a concrete edge case. A, B, and D are vague and give the model nothing precise to optimize for.

**2.** You need generated sample rows to match an exact JSON shape. Which technique fits best?

- A. Zero-shot prompting
- B. Few-shot prompting with two example rows

- C. Lowering the temperature only
- D. Opening more unrelated files

Answer

**Correct: B.** Few-shot examples pin the format. Zero-shot (A) leaves the shape to chance; C and D don't reliably control structure.

**3.** A first Copilot Chat answer is close but misses null handling. What is the best next step?

- A. Start a new, unrelated chat and retype the whole request.
- B. In the same thread, ask "now handle null and empty inputs, and add a test."
- C. Switch to a different programming language.
- D. Repeat the original prompt verbatim.

Answer

**Correct: B.** Iterating within the thread reuses conversation context. A discards context; C is irrelevant; D changes nothing.

**4.** In Copilot Chat, what is the purpose of referencing `#selection` or a specific `#file`?

- A. It retrains the model on that file.
- B. It explicitly adds that code to the prompt's context so the answer is grounded in it.
- C. It disables content filtering for that file.
- D. It increases the context window size.

Answer

**Correct: B.** Explicit references inject specific, relevant context. They don't retrain the model (A), affect filtering (C), or change the window size (D).

**5.** Which is the best description of prompt engineering for a developer using Copilot?

- A. Writing code to fine-tune the model on your repo.
- B. Crafting clear instructions and supplying the right context and examples to improve output.
- C. Configuring the proxy server.
- D. Training a new large language model.

Answer

**Correct: B.** Prompt engineering is about instructions, context, and examples — no model training or infrastructure work. A, C, and D describe unrelated technical activities.

---

## Further reading

- **Chapter 2 — How GitHub Copilot Works**: how context is gathered into the prompt in the first place.
- **Chapter 6 — Copilot Capabilities**: reusing prompts with prompt files and setting standards with instructions files.

  **Source**: Prompt engineering for GitHub Copilot Chat

  **Source**: Study guide for Exam GH-300: GitHub Copilot

# Chapter 4 — Using AI Responsibly

*Part I — AI foundations for developers*

---

## In 30 seconds

- **The core idea**: responsible use means understanding the risks of generative AI, validating output, and operating Copilot within ethical and organizational boundaries.
- **Why it matters**: you remain accountable for code you accept; unreviewed AI output creates security, quality, and licensing risk.
- **The exam angle**: GH-300 tests **risks and limitations**, **ethical/responsible use**, **potential harms and mitigations**, **the need to validate AI output**, and **operating Copilot responsibly**.
- **Remember**: human review is non-negotiable — Copilot assists, you decide.

---

## Exam map

**Exam map — GH-300 · Use GitHub Copilot responsibly · Foundation for GH-600 guardrails**

---

## 1. Key concepts

Copilot accelerates your work, but it does not assume responsibility for it. You do. Responsible use means understanding what can go wrong, validating what the model produces, and operating Copilot within ethical and organizational boundaries. GH-300 gives this a full skill area (15–20%), and the same instincts become the foundation for **guardrails** in the GH-600 agent track (Chapter 14).

> **Definition — Responsible AI**: the practice of designing, building, and using AI systems in ways that are fair, reliable and safe, private and secure, inclusive, transparent, and accountable. Microsoft organizes its approach around these six principles.

### The six responsible-AI principles

| Principle | What it means in practice with Copilot |
|---|---|
| **Fairness** | Watch for biased patterns learned from training data; don't let generated logic encode discrimination. |
| **Reliability & safety** | Suggestions can be wrong or unsafe; test and review before trusting them. |
| **Privacy & security** | Don't expose secrets or sensitive data; use content exclusions; verify generated code isn't insecure. |
| **Inclusiveness** | Consider diverse users and inputs the model may not represent well. |
| **Transparency** | Be clear that AI assisted; understand and explain what the tool does and its limits. |
| **Accountability** | A human remains answerable for the code that ships — Copilot is not accountable, you are. |

> **Key concept**: Copilot is an assistant, not an authority. Every suggestion is a proposal that you are responsible for reviewing, testing, and owning.

### The risks you must be able to name

- **Fabrication (hallucination)** — confident but wrong code, invented APIs or flags (Chapter 1).
- **Insecure patterns** — suggestions can include injection-prone queries, weak crypto, or missing input validation, because such patterns exist in training data.
- **Bias** — the model can reproduce biased assumptions present in its training corpus.
- **Outdated practice** — deprecated APIs or old idioms from the knowledge cutoff.
- **Intellectual property / licensing** — a suggestion may resemble public code; the duplication filter (Chapter 8) mitigates this, and you remain responsible for compliance.
- **Over-reliance** — accepting output without understanding it erodes quality and your own judgment.

---

## 2. How it works

### Harms, and how to mitigate them

Responsible use is not a vibe; it is a set of concrete mitigations mapped to concrete risks:

| Risk | Mitigation |
|---|---|
| Fabrication / wrong code | Human review; unit and integration tests; run it |
| Insecure code | Security review; static analysis / code scanning; the public-code match filter |

| Risk | Mitigation |
|---|---|
| Sensitive-data exposure | Content exclusions; never paste secrets into prompts |
| IP / licensing concerns | Enable the duplication (public-code) filter; review provenance |
| Over-reliance | Only accept code you understand; keep humans in the loop |

> **How it works**: the single most important control is **validation**. Because output is probabilistic and unverified (Chapters 1–2), you cannot delegate correctness to the model. Review, test, and scan are not optional extras — they are how AI-assisted development stays trustworthy.

### Operating Copilot responsibly, day to day

- **Understand before you accept** — if you can't explain a suggestion, don't ship it.
- **Validate output** — tests for behavior, scanning for security, review for design.
- **Protect data** — keep secrets and sensitive files out of context (exclusions), and out of prompts.
- **Be transparent** — follow your organization's norms for disclosing AI assistance.
- **Stay accountable** — the commit has your name on it.

> **Hands-on**: pair Copilot with automated checks. After accepting generated code, ask Copilot Chat to generate tests *and* run your scanner (for example, CodeQL / code scanning in CI) so validation is built into the workflow, not left to memory.

---

## 3. In the real world

**Scenario — the query that looked fine.** A developer accepts a Copilot suggestion that builds a SQL query by string-concatenating a user-supplied `search` value. It compiles, it works in the demo, and it is a textbook SQL-injection vulnerability. Nothing about the suggestion looked alarming — that is precisely the danger. A responsible workflow catches it: a security review (or a code-scanning alert) flags the concatenation, the developer switches to a parameterized query, and adds a test with a malicious input. Copilot helped write the fix, too — but a human's judgment and validation are what kept the vulnerability out of production.

---

## 4. Exam tips

> **Exam tip**: the recurring right answer is **validate the output** — review, test, and scan. Any option suggesting you can trust Copilot's code without verification is a trap.

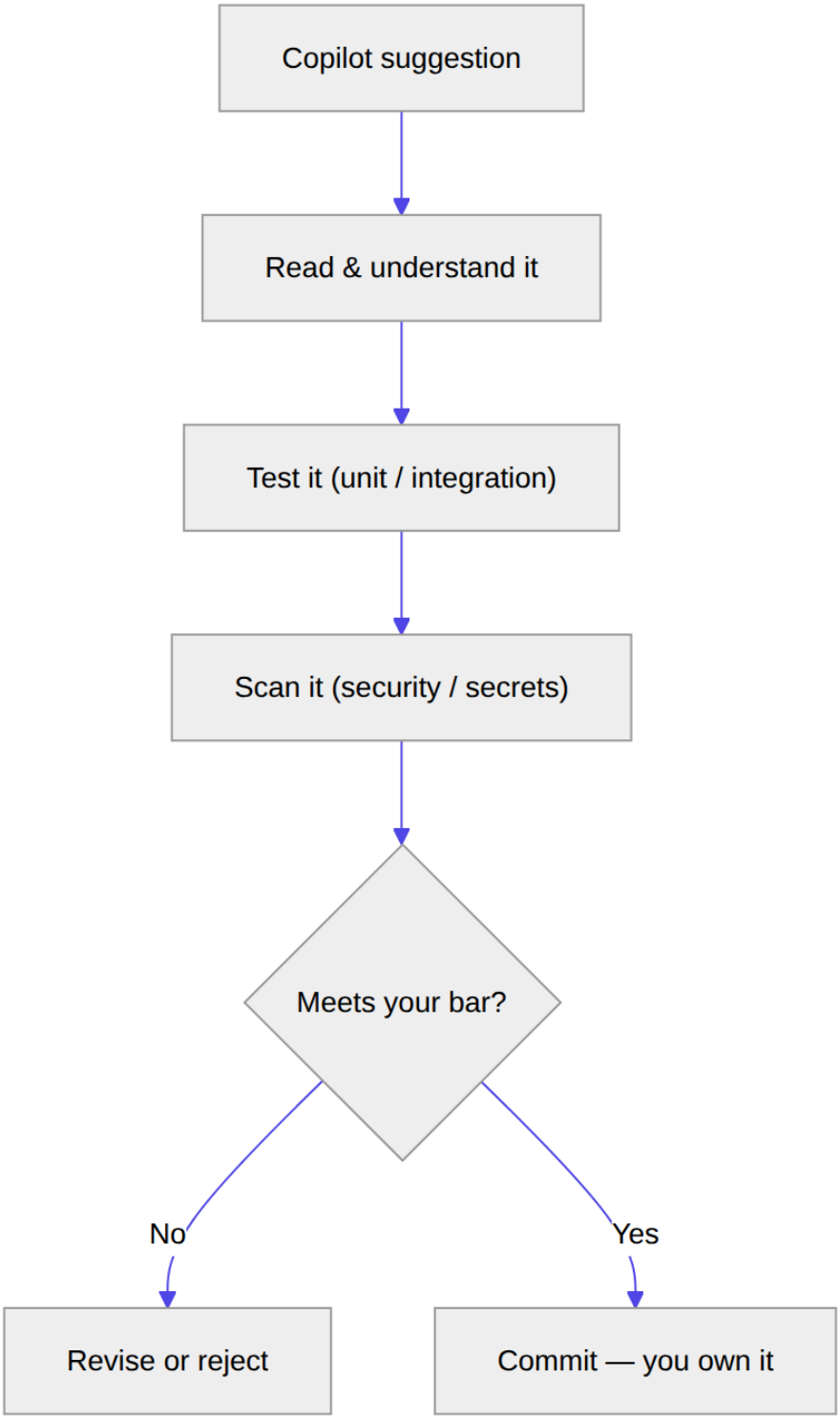


Figure 5: Diagram

**Exam tip**: be able to list **risks** (fabrication, insecure code, bias, outdated practice, IP/licensing, over-reliance) and pair each with a **mitigation**. GH-300 asks about "potential harms and mitigation strategies."

**Exam tip**: **accountability stays with the human**. Copilot assists; it is never the responsible party for shipped code.

**Exam tip**: recognize the six principles — **fairness, reliability & safety, privacy & security, inclusiveness, transparency, accountability**.

---

## 5. Common pitfalls

**Pitfall**: over-reliance. Accepting code you don't understand transfers risk to production and away from your judgment.

- **Assuming generated code is secure**: it may contain injection, weak crypto, or missing validation.
- **Assuming generated code is license-clean by default**: enable the duplication filter and review provenance; you remain responsible for compliance.
- **Pasting secrets into prompts**: sensitive data does not belong in context — use exclusions.
- **Skipping tests because "Copilot wrote it"**: authorship by AI raises the need for validation, not lowers it.

---

## 6. Practice questions

**1.** A teammate wants to merge Copilot-generated code without review "because the AI wrote it." What is the responsible position?

- A. Approve it — AI output doesn't need review.
- B. Require review, tests, and security scanning; a human is accountable for the code.
- C. Approve it only if it compiles.
- D. Reject all AI-generated code as a rule.

Answer

**Correct: B.** Validation and human accountability are the core of responsible use. A and C skip validation; D needlessly bans a useful tool rather than governing it.

**2.** Which pairing of risk and mitigation is correct?

- A. Insecure code → lower the temperature
- B. Sensitive-data exposure → content exclusions and keeping secrets out of prompts
- C. Fabrication → enable dark mode
- D. IP concerns → delete the repository

Answer

**Correct: B.** Exclusions and prompt hygiene mitigate data exposure. A, C, and D pair risks with irrelevant or absurd "mitigations."

**3.** Which is a genuine responsible-AI concern specific to code suggestions?

- A. Suggestions may reproduce insecure patterns present in training data.
- B. Suggestions are always slower than hand-written code.
- C. Suggestions cannot be edited.
- D. Suggestions disable your compiler.

Answer

**Correct: A.** Models can surface insecure patterns, so review and scanning matter. B, C, and D are false.

**4.** Under the six responsible-AI principles, which one most directly explains why *you* — not Copilot — answer for shipped code?

- A. Inclusiveness
- B. Transparency
- C. Accountability
- D. Reliability & safety

Answer

**Correct: C.** Accountability places responsibility with the human. The others are real principles but don't directly assign responsibility for outcomes.

**5.** What is the single most important step before trusting a Copilot suggestion in production?

- A. Accept it quickly to save time.
- B. Validate it — understand, test, and scan the output.
- C. Increase the context window.
- D. Share the prompt publicly.

Answer

**Correct: B.** Validation is the core mitigation for probabilistic, unverified output. A increases risk; C and D don't address correctness or safety.

---

## Further reading

- **Chapter 1 — Understanding Generative AI for Developers**: why fabrication and non-determinism make validation mandatory.
- **Chapter 8 — Administering Copilot**: content exclusions and the public-code duplication filter.
- **Chapter 14 — Guardrails & Accountability**: responsible-AI thinking applied to autonomous agents.

**Source**: Responsible AI with GitHub Copilot (Microsoft Learn)

**Source**: What is Responsible AI? (Microsoft Learn)

# Chapter 5 — Copilot in the IDE and CLI

*Part II — GH-300 track: Working with GitHub Copilot*

---

## In 30 seconds

- **The core idea**: Copilot meets you where you work — inline suggestions and chat in the IDE, and a dedicated CLI in the terminal, plus Agent Mode for multi-step tasks.
- **Why it matters**: knowing which surface to use, and how to enable it, is core day-to-day fluency.
- **The exam angle**: GH-300 tests **enabling Copilot in the IDE**, **triggering it via inline suggestions, chat, CLI, and agent mode**, and the **GitHub Copilot CLI** (install, commands, interactive sessions, script/file generation).
- **Remember**: inline = completions; chat = conversation; CLI = terminal; Agent Mode = multi-step tasks.

---

## Exam map

**Exam map — GH-300 · Use GitHub Copilot features (IDE + CLI)**

---

## 1. Key concepts

Copilot is not one feature; it is a set of surfaces you reach for depending on the job. GH-300 devotes its largest skill area (25–30%) to using those features. This chapter covers the two you live in most — the **IDE** and the **command line** — and Chapter 6 covers the advanced capabilities layered on top.

> **Key concept**: match the surface to the task. **Inline suggestions** for completing code as you type; **Copilot Chat** for a conversation about code; **agent mode** for multi-step changes across files; the **CLI** to work from your terminal.

### Enabling Copilot in the IDE

Copilot runs in editors such as Visual Studio Code, Visual Studio, JetBrains IDEs, Eclipse, Xcode, and others, via an official extension. Getting started is consistent: install the Copilot extension for your IDE, sign in with the GitHub account that has a Copilot plan, and confirm the extension is enabled. Once active, Copilot offers **inline suggestions** (ghost text you accept with Tab) and opens **Copilot Chat** in a side panel.

> **Definition — Inline suggestion**: grey "ghost text" Copilot proposes at your cursor as you type, which you accept, dismiss, or cycle through alternatives for. It is the completion experience — fast, in the flow, driven by surrounding context (Chapter 2).

### The four ways to trigger Copilot

- **Inline suggestions** — accept completions as you type.
- **Chat** — ask questions and request changes conversationally, with slash commands (`/explain`, `/fix`, `/tests`) and context references (`#file`, `#selection`).
- **CLI** — run Copilot as an agent in your terminal (below).
- **Agent mode** — let Copilot plan and apply multi-step, multi-file changes (Chapter 6).

---

## 2. How it works

### GitHub Copilot CLI: an agent in your terminal

> **Definition — GitHub Copilot CLI**: a command-line tool that brings a Copilot **agent** into your terminal. You can ask it questions, have it write and debug code, run tasks on your behalf, and interact with GitHub.com — for example, create a branch, open a pull request, or triage issues — without leaving the shell. It runs on Linux, macOS, and Windows (PowerShell or WSL).
>
> **Hands-on**: after installing the CLI, start an interactive session from a project folder and let it work with you. Verify current install steps in GitHub Docs.

```
copilot                          # start an interactive session (asks you to trust t
# then, at the prompt:
#   "Add a Node script user-info.js that prints the current user, and open a PR"
```

**Two interfaces.** The CLI has an **interactive** mode (you converse, steer, and approve actions) and a **programmatic** mode for one-shot or scripted use with `-p`/`--prompt`:

```
# Programmatic: run one task and exit
copilot -p "Show me this week's commits and summarize them" --allow-tool='shell(git)'

# Pipe options or context from a script
./script-outputting-options.sh | copilot
```

**Interactive niceties** (verify in docs, as they evolve):

- **Plan mode** — press Shift+Tab to have Copilot build a structured plan before it writes code, so you catch misunderstandings early.

- **Reference a file** with `@path/to/file` to add its contents as context.
- **Run a shell command directly** by prefixing with `!` (for example, `!git status`) — no model call.
- **Manage context** with `/context`, `/compact`, and `/usage`; sessions auto-compact near the token limit.
- **Resume** a session with `copilot --continue` or `/resume`.
- **Schedule** prompts with `/every` and `/after`.

  **How it works**: because the CLI can modify and execute files, it asks for **approval** before running tools that touch your system. You can approve per-command, for the session, or pre-authorize specific tools with flags like `--allow-tool='shell(git)'` (and deny with `--deny-tool`). `--allow-all` (or `--yolo`) removes prompts entirely — powerful and risky. Only launch the CLI from **trusted directories**.

  **Pitfall**: `--allow-all-tools` / `--yolo` gives Copilot the same access you have to run shell commands without review. Use sandboxing (`/sandbox enable`, or `copilot --cloud`) for unattended runs.

### Generating scripts and managing files

A core CLI use case is turning intent into shell work: "write a script that compresses logs older than seven days," "explain this `tar` command," "revert the last commit leaving changes unstaged," or "create a GitHub Actions workflow that runs ESLint on pull requests and fails on errors." Copilot proposes the commands or files, and — with your approval — runs or writes them.

---

## 3. In the real world

**Scenario — one task, three surfaces.** A developer is implementing a feature. She starts with **inline suggestions**, accepting completions for boilerplate. A tricky bug appears, so she selects the function and asks **Copilot Chat** `/fix #selection` for an explanation and a patch. Finally, she needs to wire up CI: rather than context-switch to the browser, she opens the **CLI** and prompts, "branch off main and add a GitHub Actions workflow that runs the tests on PRs; push it and open a pull request." The CLI plans the change, asks approval before pushing, and creates the PR with her as author — all without leaving the terminal.

---

## 4. Exam tips

**Exam tip**: know the **four triggers** — inline suggestions, chat, CLI, and agent mode — and which fits which task. "Complete this line" → inline; "explain/refactor this" → chat; "do a multi-step task in my terminal" → CLI; "apply a multi-file change" → agent mode.

**Exam tip**: the CLI is an **agent** that can act on your behalf (edit files, run commands, open PRs) and therefore asks for **approval**. Recognize the approval model and that you should run it from trusted directories.

**Exam tip**: to enable Copilot in an IDE you install the **extension** and **sign in** with an account that has a Copilot plan — no model training or server setup is involved.

---

## 5. Common pitfalls

**Pitfall**: confusing the CLI with plain chat. The modern GitHub Copilot CLI is an **agent** that can execute commands and modify files — not just answer questions.

- **Running the CLI from an untrusted or home directory**: it may read/modify files below that location. Launch from the specific project folder you trust.
- **Blanket auto-approval**: `--allow-all`/`--yolo` is convenient but removes your review step; prefer scoped `--allow-tool` or sandboxing.
- **Assuming inline = chat**: inline completes code silently; chat is conversational with commands and references.

---

## 6. Practice questions

**1.** A developer wants Copilot to create a branch, add a file, and open a pull request — all from the terminal. Which surface is designed for this?

- A. Inline suggestions
- B. GitHub Copilot CLI
- C. The syntax highlighter
- D. A `.gitignore` file

Answer

**Correct: B.** The Copilot CLI is a terminal agent that can modify files and interact with GitHub.com, including opening PRs. Inline suggestions (A) only complete code; C and D are unrelated.

**2.** What must you do to start using Copilot's inline suggestions in your IDE?

- A. Retrain the model on your repository.
- B. Install the Copilot extension and sign in with an account that has a Copilot plan.
- C. Configure a proxy server manually.
- D. Disable all other extensions.

Answer

**Correct: B.** Enabling Copilot is install-and-sign-in. A, C, and D describe things that are not required.

**3.** In the Copilot CLI, what does the `--allow-all-tools` (or `--yolo`) option do?

- A. Restricts Copilot to read-only access.
- B. Lets Copilot use any tool and run shell commands without asking for approval.
- C. Enables inline suggestions in the IDE.
- D. Doubles the context window.

Answer

**Correct: B.** It removes the per-tool approval prompts, granting broad execution power — convenient but risky. A is the opposite; C and D are unrelated.

**4.** Which CLI capability lets Copilot build a structured implementation plan before writing any code?

- A. Plan mode
- B. Inline mode
- C. The ! shell prefix
- D. The duplication filter

Answer

**Correct: A.** Plan mode has Copilot analyze the request and produce a plan first. The ! prefix (C) runs a shell command directly; B and D are unrelated.

**5.** A developer selects a buggy function and wants Copilot to explain and fix just that code in the IDE. What is the most direct approach?

- A. Accept the next inline suggestion.
- B. In Copilot Chat, use `/fix` with `#selection` to scope the request to the selected code.
- C. Reinstall the IDE.
- D. Open the CLI and run `--yolo`.

Answer

**Correct: B.** Chat with a slash command and an explicit selection reference targets exactly that code. A is unfocused; C is irrelevant; D is an unscoped, risky terminal action.

---

## Further reading

- **Chapter 6 — Copilot Capabilities**: agent mode, Copilot Edits, MCP, code review, and reuse via instructions and prompt files.
- **Chapter 3 — Prompt Engineering & Context Crafting**: slash commands and context references in chat.

  **Source**: About GitHub Copilot CLI (GitHub Docs)

  **Source**: Using GitHub Copilot CLI (GitHub Docs)

# Chapter 6 — Copilot Capabilities: Agent Mode, Edits, MCP, Code Review

*Part II — GH-300 track: Working with GitHub Copilot*

---

## In 30 seconds

- **The core idea**: beyond completions, Copilot can make multi-file edits, run as an agent with tools via MCP, review code, and reuse your standards through instructions and prompt files.
- **Why it matters**: these capabilities are where Copilot shifts from assistant to collaborator — and they are heavily represented on GH-300.
- **The exam angle**: GH-300 tests **Agent Mode, Copilot Edits, and MCP**; **Agent Sessions and sub-agents**; **code review and coding assistance**; **Spaces, Spark, PR summaries, and instructions files**; and **Chat limits, options, feedback, commands, and prompt-file reuse**.
- **Remember**: instructions files set standards; prompt files make responses repeatable; MCP adds tools.

---

## Exam map

**Exam map — GH-300 · Use GitHub Copilot features and capabilities**

---

## 1. Key concepts

Beyond completing lines, Copilot can make coordinated multi-file edits, act as an agent that uses tools, review your pull requests, and apply your team's standards automatically. GH-300 tests these under "use GitHub Copilot features and capabilities." The through-line: you move from *typing code with help* to *delegating work with oversight*.

### Agent mode vs Copilot Edits vs inline

**Definition — Copilot Edits**: an IDE capability where you describe a change and Copilot proposes edits **across multiple files at once**, which you review and accept or discard as a set. You stay in control of scope — you pick the files in context.

**Definition — Agent mode**: Copilot works **autonomously toward a goal** — it decides which files to change, runs tools (build, tests, terminal), reads the results, and iterates until the task is done or it needs you. Where Copilot Edits applies a described change, agent mode figures out *what* changes are needed and *verifies* them.

| | Inline | Copilot Edits | Agent mode |
|---|---|---|---|
| Scope | Current line/block | Multiple files you scope | Whatever the goal requires |
| Who decides what changes | You | You describe; Copilot edits | Copilot plans and edits |
| Runs tools (tests, terminal) | No | No | Yes |
| Best for | Completions | A known multi-file change | Multi-step tasks and fixes |

### Sub-agents and Agent Sessions

**Key concept**: to manage the context window (Chapter 1), Copilot can **delegate** a sub-task to a **sub-agent** running in its own context — for example, a code-exploration agent that answers a question without cluttering the main conversation. Delegating keeps the primary session focused and optimizes token usage.

### The Model Context Protocol (MCP)

**Definition — Model Context Protocol (MCP)**: an open standard for connecting AI agents to external **tools and data sources** through "MCP servers." Adding an MCP server gives Copilot new abilities — query an issue tracker, read observability data, call an internal API — in a consistent, discoverable way. Copilot comes with the **GitHub MCP server** preconfigured. (MCP is central to GH-600; here you just need to know it *extends what Copilot can do*.)

---

## 2. How it works

### Setting standards with instructions files

**Definition — Instructions file**: a Markdown file of natural-language guidance that Copilot automatically applies to requests in a repository. **Repository-wide** instructions live in `.github/copilot-instructions.md`; **path-specific** instructions live in `.github/instructions/NAME.instructions.md` with an `applyTo` glob in the frontmatter; **agent instructions** can live in `AGENTS.md` files.

**Hands-on**: create repository-wide standards Copilot will apply automatically.

```
<!-- .github/copilot-instructions.md -->
- Use TypeScript strict mode and prefer named exports.
- Write a unit test for every new function.
- Follow the existing error-handling pattern (return Result, don't throw).
```

```
<!-- .github/instructions/python.instructions.md -->
---
applyTo: "**/*.py"
---
Use type hints and docstrings. Prefer pathlib over os.path.
```

**How it works**: when several instruction sets apply, **personal** instructions take priority, then **repository**, then **organization** — but all relevant sets are provided to Copilot. Avoid contradictory guidance. For Copilot code review, instructions are read from the **head branch** of a pull request, so you can test instruction changes in the same PR.

### Reusing requests with prompt files

**Definition — Prompt file**: a reusable, parameterizable prompt saved in the repository (for example, under `.github/prompts/`) so a team runs the *same* well-crafted request the same way every time — "scaffold a REST endpoint," "write tests to our standard." Instructions files set *standing rules*; prompt files package *repeatable tasks*.

### Copilot code review

**Definition — Copilot code review**: Copilot reviews a pull request (or changes in the IDE/CLI), surfacing likely bugs, style issues, and improvements as comments — optionally guided by your instructions files so the review enforces your team's standards. It complements, not replaces, human review.

Copilot can also generate **pull request summaries**, and features such as **Spaces** (curated context you can reuse and share) and **Spark** (building an app from a description) extend where and how you work with Copilot. Confirm current availability and scope in GitHub Docs — some are evolving.

### Managing the chat conversation

Copilot Chat has **limits** (the context window), **options** (model selection, references), **commands** (`/explain`, `/fix`, `/tests`), and a **feedback** mechanism. Reusing prompt files keeps responses consistent across a team.

---

## 3. In the real world

**Scenario — a feature, shipped with standards baked in.** A team adds `.github/copilot-instructions.md` ("strict TypeScript, named exports, a test per function") and a prompt

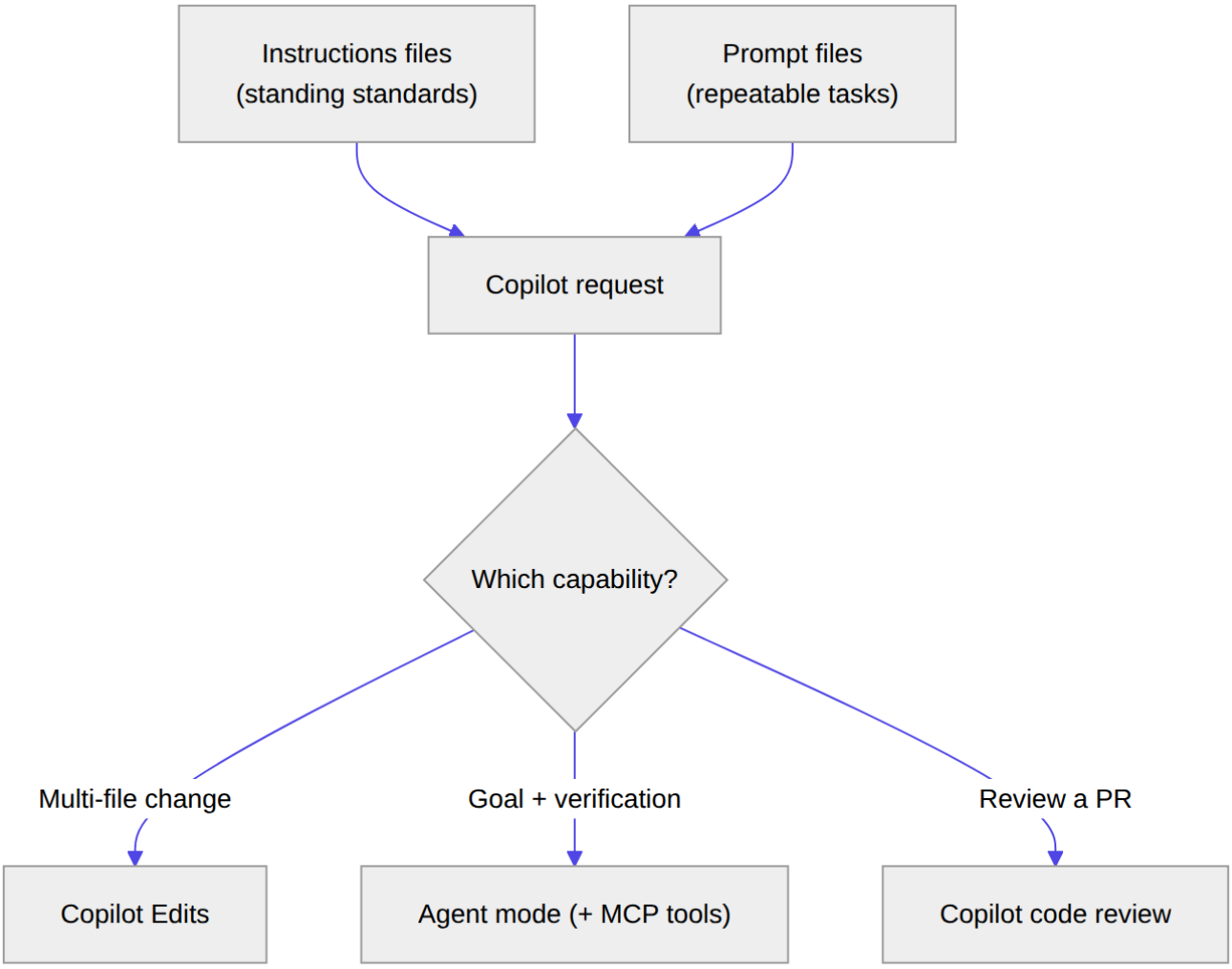


Figure 6: Diagram

file for scaffolding endpoints. A developer opens **agent mode** and says, "add a `/orders` endpoint with validation and tests." Copilot plans the change, edits the router, controller, and test files, runs the tests, fixes a failing case, and stops for review — automatically following the repo's instructions. When the pull request opens, **Copilot code review** (reading the same instructions from the head branch) flags a missing null check before a human reviewer even looks. Standards were enforced without anyone restating them.

---

## 4. Exam tips

**Exam tip**: distinguish **Copilot Edits** (you describe a change; Copilot edits multiple files you scope) from **agent mode** (Copilot decides what to change *and runs tools to verify*). "Runs the tests and iterates" points to agent mode.

**Exam tip**: **instructions files** set standing standards (`.github/copilot-instructions.md`, path-specific `*.instructions.md`, `AGENTS.md`); **prompt files** package repeatable requests. Don't swap the two.

**Exam tip**: **MCP** *extends* Copilot with external tools/data. The GitHub MCP server is built in.

**Exam tip**: for Copilot code review, custom instructions are read from the pull request's **head branch**, and code review must have custom instructions enabled (the default).

---

## 5. Common pitfalls

**Pitfall**: writing contradictory instructions across personal, repository, and organization scopes. All apply; conflicts degrade quality. Keep them coherent.

- **Confusing instructions files with prompt files**: rules vs repeatable tasks.
- **Expecting agent mode to be "just bigger autocomplete"**: it plans, edits across files, and runs tools — review its work.
- **Assuming Copilot code review replaces human review**: it augments it; accountability stays human (Chapter 4).
- **Forgetting the head-branch rule**: code-review instructions come from the PR's head branch, not base.

---

## 6. Practice questions

**1.** A developer says, "implement this feature, run the tests, and fix anything that fails." Which capability is designed for that?

- A. Inline suggestions
- B. Copilot Edits
- C. Agent mode

- D. A prompt file alone

Answer

**Correct: C.** Agent mode plans, edits across files, and runs tools (like tests), iterating on failures. Inline (A) completes code; Copilot Edits (B) applies a described multi-file change but doesn't run tools; a prompt file (D) is just a reusable request.

**2.** Where do repository-wide custom instructions for Copilot live?

- A. `.github/copilot-instructions.md`
- B. `README.md`
- C. `.gitignore`
- D. `package.json`

Answer

**Correct: A.** Repository-wide instructions are in `.github/copilot-instructions.md`; path-specific ones live under `.github/instructions/`. The others are unrelated files.

**3.** What is the purpose of adding an MCP server to Copilot?

- A. To retrain the model on your repository.
- B. To extend Copilot with external tools and data sources through a standard protocol.
- C. To disable the content filters.
- D. To increase your Copilot license count.

Answer

**Correct: B.** MCP connects agents to tools/data in a standard way. A, C, and D describe unrelated things.

**4.** When multiple sets of custom instructions apply to a request, which takes priority?

- A. Organization, then repository, then personal
- B. Personal, then repository, then organization
- C. Only the repository set is used
- D. Only the organization set is used

Answer

**Correct: B.** Personal instructions take highest priority, then repository, then organization — though all relevant sets are provided to Copilot. C and D are false; A reverses the order.

**5.** What best distinguishes an instructions file from a prompt file?

- A. They are identical.
- B. Instructions files set standing standards applied automatically; prompt files package repeatable requests.
- C. Prompt files disable agent mode; instructions files enable it.
- D. Instructions files only work in the CLI.

Answer

**Correct: B.** Instructions = standing rules; prompt files = reusable tasks. A, C, and D are incorrect.

---

## Further reading

- **Chapter 5 — Copilot in the IDE and CLI**: the surfaces these capabilities build on.
- **Chapter 10 — Tool Use & Environment Interaction (MCP)**: MCP servers, registries, and allow lists in depth (GH-600).

  **Source**: Adding repository custom instructions for GitHub Copilot

  **Source**: Extending Copilot with the Model Context Protocol (MCP)

# Chapter 7 — Boosting Developer Productivity

*Part II — GH-300 track: Working with GitHub Copilot*

---

## In 30 seconds

- **The core idea**: Copilot accelerates the everyday loop — generating, refactoring, documenting, testing, and hardening code — while you steer and verify.
- **Why it matters**: productivity is the whole point of the tool, and GH-300 tests concrete use cases.
- **The exam angle**: GH-300 tests **code generation, refactoring, documentation**; **accelerating learning and reducing context switching**; **sample data and legacy modernization**; **unit/integration tests, edge cases, assertions**; and **security improvements and performance optimizations**.
- **Remember**: Copilot drafts; tests and review confirm.

---

## Exam map

**Exam map — GH-300 · Improve developer productivity with GitHub Copilot**

---

## 1. Key concepts

Everything so far — how the model works, how to prompt it, which surface to use — exists to make you faster and your code better. GH-300 tests concrete productivity use cases directly, and they cluster into a simple loop: Copilot **drafts**, you **verify**. This chapter is a tour of where that loop pays off across the development lifecycle.

> **Key concept**: Copilot's productivity gains are real but conditional. They show up when you use it for the right tasks — boilerplate, tests, docs, translation, exploration — and keep a human validating the output (Chapter 4). Speed without verification is just faster bugs.

### Where Copilot helps most

- **Code generation** — scaffolding, boilerplate, well-known algorithms, repetitive patterns.
- **Refactoring** — extracting functions, renaming, restructuring, applying a pattern consistently.
- **Documentation** — docstrings, README sections, inline comments, usage examples.
- **Learning & reduced context switching** — explaining unfamiliar code or APIs *in your editor* instead of a web search.
- **Sample data** — realistic fixtures and seed data in the exact shape you need.
- **Legacy modernization** — translating idioms, upgrading patterns, explaining old code before you change it.

> **Definition — Context switching**: the productivity cost of leaving your task to look something up elsewhere. By answering questions and generating examples in place, Copilot reduces these switches — a significant, if hard-to-measure, gain.

---

## 2. How it works

### Testing: generation, edge cases, and assertions

Test support is one of Copilot's highest-value use cases and a named GH-300 objective. Copilot can generate **unit and integration tests**, propose **edge cases** you might miss (empty input, boundaries, duplicates, nulls, concurrency), and write **assertions** that check the right behavior.

> **Hands-on**: scope the request and ask for the cases that matter.

```
/tests #selection
Generate unit tests for the selected function. Include: empty input, a single ele
a very large input, and a malformed record. Use the project's existing test frame
and assert both the return value and that no exception is thrown for valid input.
```

> **Pitfall**: a generated test that *passes* is not automatically a *good* test. Confirm it asserts the intended behavior — Copilot can write a test that locks in a bug.

### Security and performance suggestions

Copilot can **suggest security improvements** (parameterized queries instead of string concatenation, input validation, safer defaults) and **performance optimizations** (removing needless work in a loop, better data structures). Treat these as *proposals* to validate with review and scanning (Chapter 4), not guarantees.

### Generation, refactoring, and documentation in practice

> **How it works**: the win comes from *drafting speed*. Producing a first version of tests, a docstring, or a refactor is where most time is spent; Copilot collapses that to seconds, leaving you to do the higher-value work of judging and correcting it.

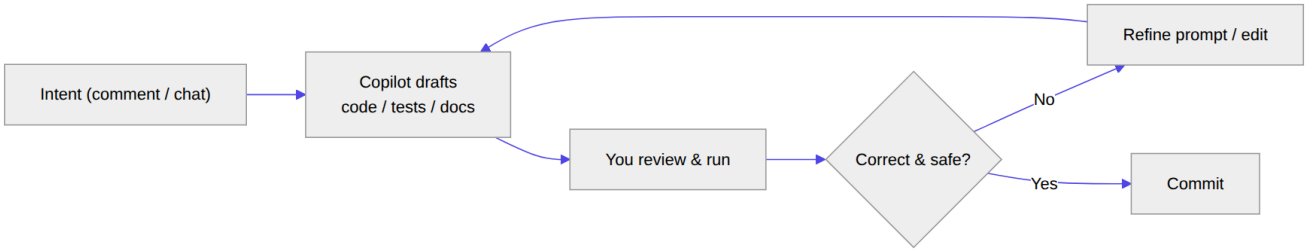


Figure 7: Diagram

### Generating sample data and modernizing legacy code

Need 50 realistic user records as JSON matching a schema? Ask, with two examples (few-shot, Chapter 3). Facing a legacy module? Ask Copilot to **explain** it first, then to **modernize** a piece at a time — with tests around each change so you can trust the transformation.

---

## 3. In the real world

**Scenario — a Friday-afternoon refactor.** A developer must extract validation logic duplicated across three handlers, add tests, and document the new helper. She asks Copilot to **explain** the current duplication, uses **Copilot Edits** to extract a shared `validateOrder` function across the files, then prompts `/tests` for the helper — explicitly requesting empty-input and boundary cases. Copilot drafts the tests; two fail, revealing a real edge case the old code mishandled. She fixes the helper, asks for a docstring, and opens a pull request. What would have been an hour of tedium is fifteen minutes of judgment — because Copilot did the drafting and she did the verifying.

---

## 4. Exam tips

**Exam tip**: know the productivity use cases by name — **generation, refactoring, documentation, tests (with edge cases and assertions), sample data, legacy modernization, security and performance suggestions**. GH-300 asks which task Copilot is well suited to.

**Exam tip**: the correct framing is always **draft then verify**. Any answer implying Copilot's tests or security fixes can be trusted without review is wrong.

**Exam tip**: "reducing context switching" and "accelerating learning" are legitimate, tested benefits — explaining code in the editor counts.

---

## 5. Common pitfalls

**Pitfall**: trusting a green test suite Copilot wrote without checking that the assertions are meaningful. Passing ≠ correct.

- **Treating performance/security suggestions as guarantees**: validate with scanning and benchmarks.
- **Over-generating**: asking for a huge change in one shot buries errors; work in reviewable steps.
- **Skipping the explanation step on legacy code**: understand before you modernize.
- **Ignoring the edge cases**: the value of Copilot's test help is largely in the cases you'd have missed — ask for them explicitly.

---

## 6. Practice questions

**1.** Which is the most appropriate, well-suited task for GitHub Copilot?

- A. Guaranteeing that a function is free of all security vulnerabilities.
- B. Generating unit tests and proposing edge cases for a function, which you then review.
- C. Deciding your product roadmap.
- D. Certifying code as production-ready without human review.

Answer

**Correct: B.** Test generation with edge-case suggestions is a core, well-suited use case — followed by review. A and D overclaim guarantees; C is outside the tool's role.

**2.** A generated unit test passes on the first run. What should you do before relying on it?

- A. Nothing — a passing test is always correct.
- B. Verify the test actually asserts the intended behavior and covers meaningful cases.
- C. Delete your other tests.
- D. Increase the temperature.

Answer

**Correct: B.** A passing test can still assert the wrong thing (even lock in a bug). A is unsafe; C and D are irrelevant.

**3.** How does Copilot most directly reduce context switching?

- A. By training a model on your browser history.
- B. By explaining unfamiliar code and generating examples in the editor, so you don't leave to search.
- C. By blocking access to the internet.
- D. By disabling other extensions.

Answer

**Correct: B.** Answering questions and producing examples in place keeps you in flow. A, C, and D don't describe how Copilot works.

**4.** You need to modernize a poorly understood legacy module. What is the best first step with Copilot?

- A. Ask it to rewrite the whole module at once with no tests.
- B. Ask it to explain the module, then modernize incrementally with tests around each change.
- C. Delete the module and hope for the best.

- D. Ask it to guarantee the rewrite is bug-free.

Answer

**Correct: B.** Understand first, then change in small, verified steps. A is risky; C is reckless; D expects a guarantee Copilot can't give.

**5.** Copilot suggests replacing a string-concatenated SQL query with a parameterized one. How should you treat this?

- A. As a guaranteed fix requiring no further checks.
- B. As a helpful security *suggestion* to review, test, and scan before shipping.
- C. As irrelevant to security.
- D. As a reason to skip code review.

Answer

**Correct: B.** It's a good suggestion, but it's still validated like any change. A and D drop verification; C is wrong — it's a genuine security improvement.

---

## Further reading

- **Chapter 4 — Using AI Responsibly**: why every productivity gain is paired with validation.
- **Chapter 6 — Copilot Capabilities**: agent mode and Copilot Edits for larger, tool-verified changes.

  **Source**: Develop unit tests using GitHub Copilot tools (Microsoft Learn)

  **Source**: Developer use cases for AI with GitHub Copilot (Microsoft Learn)

# Chapter 8 — Administering Copilot: Plans, Policies & Safeguards

*Part II — GH-300 track: Working with GitHub Copilot*

---

## In 30 seconds

- **The core idea**: organizations control Copilot through policies, audit logs, subscription management, content exclusions, and safeguards like the public-code matching filter.
- **Why it matters**: admins decide where Copilot runs, what it can see, and how outputs are governed.
- **The exam angle**: GH-300 tests **organization-wide policy management**, **Copilot Code Review policies**, **feature availability across IDEs and github.com**, **audit log events**, **subscription management via REST API**, plus **content exclusions**, **output ownership/limitations**, **public-code filtering**, and **troubleshooting**.
- **Remember**: content exclusions limit what Copilot sees; the duplication filter limits what it emits.

---

## Exam map

**Exam map — GH-300 · Manage organization-wide settings and policies · Configure privacy, content exclusions, and safeguards**

---

## 1. Key concepts

Using Copilot well is one thing; **governing** it across an organization is another, and GH-300 tests both. Administrators decide where Copilot runs, which features are available, what it is allowed to see, and how its use is audited. Two safeguards recur throughout and are easy to confuse — keep them straight and much of this chapter falls into place.

> **Key concept**: the two safeguards act on **opposite ends** of the flow (Chapter 2). **Content exclusions** control what Copilot can *see* (input side). The **duplication**

**detection filter** controls what Copilot is allowed to *show* when output matches public code (output side).

**Definition — Content exclusion**: an administrator setting that prevents specified files or repositories from being used as context by Copilot — they are never sent to the model. Used to keep secrets, sensitive data, or certain paths out of prompts.

**Definition — Duplication detection filter (public-code match)**: an optional setting that suppresses suggestions matching public code on GitHub. It reduces the chance of surfacing code that resembles existing public repositories — an IP/licensing safeguard.

### Plans and where features apply

Copilot comes in plans (for example, individual, Business, and Enterprise) that differ in features, data handling (Chapter 2), and administrative control. Administrators manage **feature availability** — which capabilities are enabled — across **IDEs** and **github.com**, so an organization can, say, allow chat but gate a Preview feature.

---

## 2. How it works

### Policies and feature management

Organization and enterprise owners set **policies** that enable or disable Copilot capabilities for members — including whether **Copilot code review** is available, whether certain features run in IDEs versus on github.com, and whether the duplication filter is on. Policies cascade from enterprise to organization, giving central control with local flexibility.

**How it works**: policy management is about *availability*, content exclusions are about *visibility*, and the duplication filter is about *output*. A question that asks "how do I stop Copilot from reading `/secrets`?" is a **content exclusion**; "how do I stop suggestions that copy public code?" is the **duplication filter**; "how do I turn code review off for the org?" is a **policy**.

### Auditing use

**Definition — Audit log events**: records of Copilot-related administrative and usage events (policy changes, feature toggles, and more) available to owners for oversight and compliance. Use them to answer "who changed what, and when?"

### Managing subscriptions with the REST API

Beyond the UI, GitHub exposes a **Copilot REST API** to manage seats/subscriptions programmatically — add or remove users, list assignments, and pull usage — which scales administration for large orgs.

**Hands-on**: configure a content exclusion so Copilot never sees sensitive paths. Exclusions are defined in repository or organization settings; the schema is YAML-like. Verify current syntax in GitHub Docs.

```
# Exclude paths from Copilot context (illustrative — confirm current schema)
"*":
  - "/secrets/**"
  - "**/*.env"
```

### The public-code match filter and output ownership

Enabling the duplication filter suppresses suggestions that match public code. Separately, understand **output ownership and limitations**: under GitHub's terms you generally own the suggestions you accept, but you remain responsible for validating and for license compliance (Chapter 4). The filter reduces risk; it does not transfer responsibility.

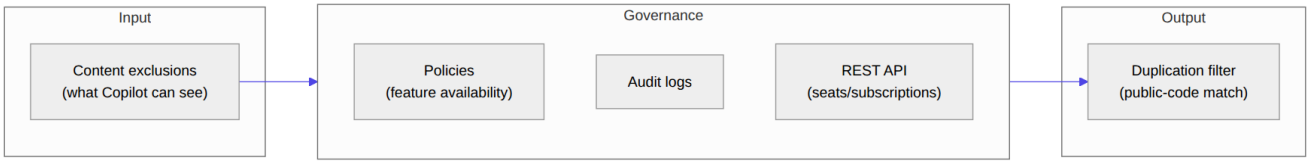


Figure 8: Diagram

### Troubleshooting

When suggestions don't appear or exclusions seem ineffective, check the usual causes: the user's plan and seat assignment, whether a **policy** disables the feature, whether a **content exclusion** is (correctly) suppressing context, network/auth issues, and IDE extension status. Exclusions apply going forward and depend on correct path patterns.

---

## 3. In the real world

**Scenario — rolling Copilot out to a regulated team.** A platform admin enables Copilot Business for the org. She sets a **policy** allowing chat and code review but disabling a Preview feature until it's GA. She adds **content exclusions** for the `secrets/` folder and all `*.env` files so credentials never reach the model. She turns on the **duplication filter** to reduce public-code matches. She scripts seat assignment with the **REST API** so onboarding is automatic, and she reviews **audit logs** weekly to confirm no policy drift. Six controls, each mapped to a specific concern — visibility, availability, output, scale, and oversight.

---

## 4. Exam tips

> **Exam tip**: the most common trap is swapping the two safeguards. **Content exclusions = input (what Copilot sees)**; **duplication filter = output (what it shows when matching public code)**.
>
> **Exam tip**: map the control to the goal — **policies** enable/disable features and set availability across IDEs and github.com; **audit logs** answer "who did what"; the **REST API** manages seats/subscriptions at scale.

**Exam tip**: you generally **own accepted suggestions**, but you remain responsible for validation and licensing. The duplication filter mitigates risk; it doesn't remove your responsibility.

**Exam tip**: content exclusions apply **going forward** and depend on correct **path patterns** — a common troubleshooting cause.

---

## 5. Common pitfalls

**Pitfall**: assuming content exclusions retroactively purge data already processed. They stop future context; they are not a delete button.

- **Confusing exclusions with the duplication filter**: input visibility vs output matching.
- **Expecting policies to filter content**: policies govern *feature availability*, not what files are seen.
- **Wrong glob in an exclusion**: a mistyped path pattern silently fails to exclude — verify it.
- **Treating the duplication filter as a licensing guarantee**: it reduces matches; you still validate compliance.

---

## 6. Practice questions

**1.** An admin wants to ensure Copilot never uses files under `secrets/` as context. Which control?

- A. The duplication detection filter
- B. Content exclusions
- C. An audit log event
- D. A prompt file

Answer

**Correct: B.** Content exclusions keep specified paths out of the model's context. The duplication filter (A) acts on outputs; C is a record; D is a reusable prompt.

**2.** Which control reduces the chance of Copilot showing suggestions that match public code?

- A. Content exclusions
- B. The duplication detection (public-code match) filter
- C. The REST API
- D. Audit logs

Answer

**Correct: B.** The duplication filter suppresses public-code matches (output side). A controls input; C manages seats; D records events.

**3.** An organization wants to add and remove Copilot seats automatically as part of onboarding. What should they use?

- A. The Copilot REST API for subscription/seat management
- B. Content exclusions
- C. The duplication filter
- D. A `.github/copilot-instructions.md` file

Answer

**Correct: A.** The REST API manages seats/subscriptions programmatically. B and C are safeguards; D sets coding standards.

**4.** A developer reports Copilot suddenly stopped suggesting in a certain repository. Which is a plausible *governance* cause to check first?

- A. The monitor refresh rate
- B. An organization policy disabling the feature, or a content exclusion covering those files
- C. The color theme
- D. The model's temperature

Answer

**Correct: B.** Policies (availability) and content exclusions (visibility) are the first governance causes to check. A, C, and D are unrelated.

**5.** Which statement about accepted Copilot suggestions is correct?

- A. GitHub owns all suggestions you accept.
- B. You generally own accepted suggestions but remain responsible for validation and license compliance.
- C. Accepting a suggestion transfers all legal responsibility to GitHub.
- D. The duplication filter guarantees license compliance.

Answer

**Correct: B.** You generally own accepted output and stay responsible for it. A and C misstate ownership; D overclaims the filter's role.

---

## Further reading

- **Chapter 2 — How GitHub Copilot Works**: why exclusions act on input and the match filter acts on output.
- **Chapter 4 — Using AI Responsibly**: validation and license responsibility that governance supports.

  **Source**: Configuring and auditing content exclusion for GitHub Copilot

  **Source**: Managing policies and features for Copilot in your organization

# Chapter 9 — Agent Architecture & SDLC Integration

*Part III — GH-600 track: Developing agentic AI systems*

---

## In 30 seconds

- **The core idea**: a well-designed agent has clear steps, defined inputs/outputs/success criteria, a planning phase separate from action, and inspectable artifacts you can supervise.
- **Why it matters**: architecture decisions determine whether an agent is safe, reviewable, and useful in a real SDLC.
- **The exam angle**: GH-600 tests **identifying agent steps and anti-patterns**, **defining inputs/outputs/success criteria**, **separating planning from execution**, **structured plans and validation**, and **observability, autonomy, artifacts, and human intervention**.
- **Remember**: plan first, validate the plan, then act — with an audit trail.

---

## Exam map

**Exam map — GH-600 · Prepare agent architecture and SDLC processes**

---

## 1. Key concepts

Welcome to the GH-600 track. Part II was about *using* Copilot; from here on, you are *engineering* agents — autonomous systems that plan, act, and produce work inside the software development lifecycle (SDLC), with GitHub as the system of record. This first chapter is about **architecture**: what makes an agent reliable, reviewable, and safe to run. Get the shape right and everything downstream — tools, memory, evaluation, guardrails — becomes tractable.

> **Definition — Agent**: an AI system that pursues a goal by **planning** a sequence of steps, **taking actions** through tools, observing the results, and iterating —

with some degree of autonomy — rather than producing a single response. In the GitHub context, an agent operates on repositories, issues, pull requests, and CI.

### Define the job before the agent

An agent is only as good as its specification. Before wiring anything, define three things:

- **Inputs** — what the agent receives (an issue, a failing test, a spec, a repository scope).
- **Outputs** — what "done" produces (a pull request, a report, a fix, an artifact).
- **Success criteria** — how you *know* it succeeded (tests pass, the check is green, the plan is approved).

**Key concept**: an agent without explicit success criteria cannot be evaluated (Chapter 12) or trusted. "Make it better" is not a spec; "open a PR that fixes issue #123 with a passing regression test" is.

### Separate planning from action

**Definition — Structured plan**: an explicit, inspectable list of the steps an agent intends to take, produced *before* it acts. Separating **planning/reasoning** from **execution** lets a human (or a policy) validate the plan and stop the agent before any change is made.

**How it works**: the pattern is *plan → validate → act*. Configure the agent to output a structured plan, validate it (automatically or with a human), and prevent action until the plan is approved. This is the single most important architectural decision for safety — it turns an opaque "it just did things" into a reviewable proposal. The Copilot CLI's **plan mode** is a concrete example.

---

## 2. How it works

### Anti-patterns to design out

| Anti-pattern | Why it hurts | Fix |
|---|---|---|
| **No plan** — agent acts immediately | Unreviewable, unsafe changes | Plan → validate → act |
| **Unbounded scope** | Touches unrelated code; huge blast radius | Scope to a repo/branch/task (Chapter 10) |
| **No success criteria** | Can't tell success from failure | Define measurable outcomes |
| **No artifacts** | Nothing to review or audit | Emit inspectable artifacts (plan, logs, diff, PR) |
| **All-or-nothing autonomy** | Either micromanaged or dangerous | Right-size autonomy with guardrails (Chapter 14) |

### Observability and control by design

> **Definition — Inspectable artifact**: a durable, reviewable output an agent produces as it works — a structured plan, a session log, a diff, a draft pull request. Artifacts are how humans supervise an agent without watching every token.

GitHub's own coding agent illustrates the pattern: it works on a branch, opens a **draft pull request**, writes **session logs** linked from each commit, and signs its commits so they appear "Verified." The change is never invisible — it lands as a reviewable artifact in the normal SDLC.

> **Hands-on**: give an agent a durable specification it can plan against — an issue plus repository instructions. A well-written `.github/copilot-instructions.md` (Chapter 6) that documents build, test, and validation steps measurably improves an agent's plans and reduces failed runs.

### Right-sizing autonomy

Autonomy is a dial, not a switch. Plan how much the agent may do unattended, and where a human must intervene — **without** turning every step into an approval that kills delivery speed (Chapter 14 formalizes this). The goal: inspectable artifacts and intervention points at the moments that matter (irreversible or risky actions), automation everywhere else.

---

## 3. In the real world

**Scenario — an agent that fixes flaky tests.** A team wants an agent to fix intermittently failing tests. Instead of "go fix the tests," they specify it properly: **input** = a labeled issue linking the flaky test; **output** = a draft PR on a `copilot/` branch; **success criteria** = the test passes 100 times in CI and no other test regresses. They configure **plan mode** so the agent first proposes *which* tests it will touch and *how* — a plan a human approves in seconds. The agent executes, emits a diff and session log, opens a draft PR, and CI runs only after a maintainer approves the workflows. Because the architecture put planning, scope, success criteria, and artifacts up front, the team supervises by exception — reading a plan and a PR, not babysitting the model.

---

## 4. Exam tips

> **Exam tip**: the core pattern is **plan → validate → act**. Configuring an agent to **output a structured plan** and **prevent action until approved** is the textbook safe design.

> **Exam tip**: every agent needs **defined inputs, outputs, and success criteria**. Questions that describe an agent with no measurable success condition are describing an anti-pattern.

> **Exam tip**: **inspectable artifacts** (plans, logs, diffs, draft PRs) are how you get observability and human intervention *without* slowing delivery. Recognize artifacts as the supervision mechanism.

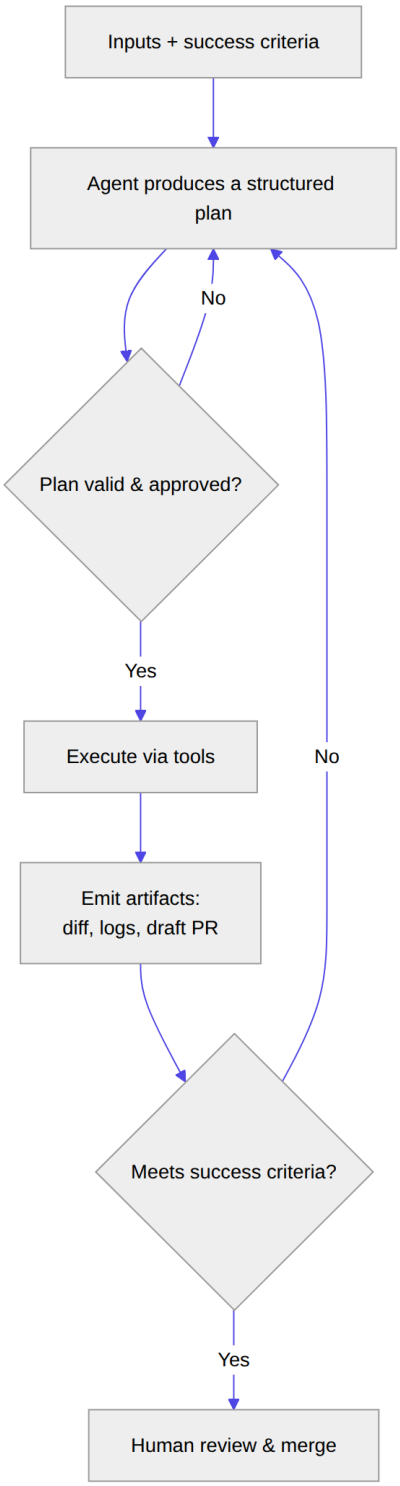


Figure 9: Diagram

**Exam tip**: know the common **anti-patterns** — acting without a plan, unbounded scope, no success criteria, no artifacts.

---

## 5. Common pitfalls

**Pitfall**: letting an agent act before it plans. Without a validated plan, you lose the one checkpoint where a human can cheaply prevent a bad change.

- **Vague success criteria**: if you can't test "done," you can't trust or evaluate the agent.
- **Unbounded scope**: an agent allowed to touch anything will eventually touch the wrong thing.
- **No artifacts**: work that leaves no plan, log, or diff can't be reviewed or audited.
- **Autonomy as a switch**: full autonomy or full micromanagement — both fail; right-size per action risk.

---

## 6. Practice questions

**1.** What is the safest architectural pattern for an agent that changes code?

- A. Act first, then explain afterward.
- B. Produce a structured plan, validate/approve it, then execute.
- C. Give it full autonomy with no plan to maximize speed.
- D. Require a human to type every command.

Answer

**Correct: B.** Plan → validate → act provides a cheap, human-reviewable checkpoint before any change. A removes the checkpoint; C is unsafe; D defeats the purpose of an agent.

**2.** Which trio must you define before building an agent?

- A. Colors, fonts, and layout
- B. Inputs, outputs, and success criteria
- C. Temperature, model, and seed
- D. Branch name, commit hash, and tag

Answer

**Correct: B.** Inputs, outputs, and measurable success criteria define the agent's job. The others are incidental or unrelated.

**3.** How do you supervise an autonomous agent without watching every action?

- A. Disable logging to reduce noise.
- B. Rely on inspectable artifacts — plans, session logs, diffs, and draft pull requests.
- C. Trust it completely.
- D. Run it only on Fridays.

Answer

**Correct: B.** Artifacts make the agent's work reviewable and auditable. A hides information; C abandons oversight; D is irrelevant.

**4.** Which is an agent anti-pattern?

- A. Scoping the agent to a specific repository and branch
- B. Emitting a structured plan before acting
- C. Acting immediately with unbounded scope and no success criteria
- D. Producing a draft pull request for review

Answer

**Correct: C.** Immediate, unbounded, unmeasurable action is the classic anti-pattern. A, B, and D are good practices.

**5.** Why configure planning to be distinct from execution?

- A. It makes the agent slower on purpose.
- B. It lets a human or policy validate the plan and stop the agent before any change is made.
- C. It disables the agent's tools permanently.
- D. It removes the need for success criteria.

Answer

**Correct: B.** Separating plan from action creates a validation checkpoint. A, C, and D misstate the purpose.

---

## Further reading

- **Chapter 10 — Tool Use & Environment Interaction (MCP)**: scoping and safe execution for agent actions.
- **Chapter 12 — Evaluation, Error Analysis & Tuning**: turning success criteria into evaluation signals.
- **Chapter 14 — Guardrails & Accountability**: right-sizing autonomy per action risk.

  **Source**: Study guide for Exam GH-600: Developing in Agentic AI Systems

  **Source**: Risks and mitigations for GitHub Copilot cloud agent

# Chapter 10 — Tool Use & Environment Interaction (MCP)

*Part III — GH-600 track: Developing agentic AI systems*

---

## In 30 seconds

- **The core idea**: agents act through tools; MCP is the standard way to expose those tools, and scoping, permissions, and safe execution paths keep the agent contained.
- **Why it matters**: tool configuration is the largest GH-600 skill area (20–25%) and where most real risk lives.
- **The exam angle**: GH-600 tests **selecting/configuring tools and permissions**, **MCP servers (remote GitHub MCP, registries, allow lists)**, **execution context, repo/branch scoping, CI invocation, autonomous PRs, environment constraints**, and **safe execution (error handling, retries, rollbacks, escalation, traceability)**.
- **Remember**: least privilege on every tool; contain scope to a repo/branch.

---

## Exam map

**Exam map — GH-600 · Implement tool use and environment interaction**

---

## 1. Key concepts

An agent that can only think is useless; an agent that can *act* is powerful — and risky. Tools are how an agent acts, MCP is the standard way to expose them, and scoping plus permissions are how you keep the agent contained. This is the **largest GH-600 skill area (20–25%)** and where most real-world risk concentrates.

> **Definition — Tool**: a capability an agent can invoke to affect the world — read a file, run a command, query an API, open a pull request. Each tool is a potential action *and* a potential risk, so tools carry **permissions**.

**Definition — MCP server**: a server implementing the **Model Context Protocol** that exposes a set of tools (and data) to an agent in a standard, discoverable way. Copilot ships with the **GitHub MCP server** preconfigured; you add others (remote HTTP or local) to extend what the agent can do.

### Select the right tools — and only those

**Key concept**: apply **least privilege** to tools. Give the agent exactly the tools the task requires, with the narrowest permissions, and no more. An agent that only needs to read issues should not have a tool that can force-push.

---

## 2. How it works

### Configuring MCP servers

You add MCP servers to give an agent new tools. With the Copilot CLI (Chapter 5), you can add a remote server directly or interactively:

**Hands-on**: add a remote MCP server as a tool (verify current syntax in GitHub Docs).

```
copilot mcp add --transport http sentry https://mcp.sentry.dev/mcp
# or, inside an interactive session:
/mcp add
```

A JSON configuration describes each server (name, transport, URL/command). For example:

```
{
  "servers": {
    "github": { "type": "http", "url": "https://api.githubcopilot.com/mcp/" }
  }
}
```

### Governing MCP: allow lists and registries

At organization/enterprise scale, you control MCP centrally. Two mechanisms matter for the exam:

- The **“MCP servers in Copilot” policy** decides whether MCP servers may run at all. Recommended: keep it enabled and restrict to an approved list.
- **Allow lists** define which servers are permitted. The **recommended** method is an enterprise **`managed-settings.json`** file (generally available; secure matching by **name, URL, or `stdio` command**; users can’t override it). An alternative is hosting your **own MCP registry** (public preview; weaker enforcement — matching by name/ID only, and users can bypass it by editing config).

  **Definition — MCP allow list**: an administrator-defined list of approved MCP servers. Prefer `managed-settings.json` for strong, non-overridable enforcement over a custom registry.

> **Exam tip**: for *strong* MCP enforcement, the answer is the **managed settings file**, not a custom registry. The registry is weaker and bypassable.

### Scoping the execution environment

Where and how an agent runs is as important as which tools it has. Evaluate the **execution context** and contain it:

- **Repository scope** — restrict the agent to a specific repository.
- **Branch scope** — GitHub's cloud agent pushes only to a **single branch**: a new `copilot/` branch, or the PR branch when triggered via `@copilot` on an existing PR. It is subject to **branch protections and required checks**.
- **CI invocation** — an agent can be invoked from a workflow, but by default **GitHub Actions workflows do not run** on the agent's PR until a user with write access approves them.
- **Autonomous branches/PRs** — the agent can create branches and open **draft** pull requests, but cannot mark them "Ready for review," approve, or merge.
- **Environment constraints** — restricted internet access (firewall) limits exfiltration; limited credentials mean it can only perform simple push operations, not arbitrary `git`.

### Safe execution paths

Actions fail; robust agents plan for it. Build in **error handling**, **retries** (with limits, so a loop doesn't run forever), **rollbacks** (undo partial changes), **escalation paths** (hand off to a human when stuck), and **traceability** (log every action for accountability, Chapter 14). GitHub's coding agent makes its actions traceable via **signed commits**, **session logs**, and **audit log events**.

---

## 3. In the real world

**Scenario — an agent that triages incidents.** A platform team builds an agent that, given an alert, reads related errors and opens a fix PR. It needs an **observability MCP server** (read-only) plus the GitHub MCP server. The admin adds the observability server to the enterprise **`managed-settings.json`** allow list so no one can wire up an unapproved one. The agent is **scoped** to the service's repository, pushes only to a `copilot/` branch, and its workflows require a maintainer's approval before running. When a tool call to the observability API times out, the agent **retries** twice, then **escalates** by commenting on the issue for a human. Every action lands in the session log. Powerful automation — tightly contained.

---

## 4. Exam tips

> **Exam tip**: **least privilege** on tools and **scoping** to a repo/branch are the recurring right answers for containing an agent.

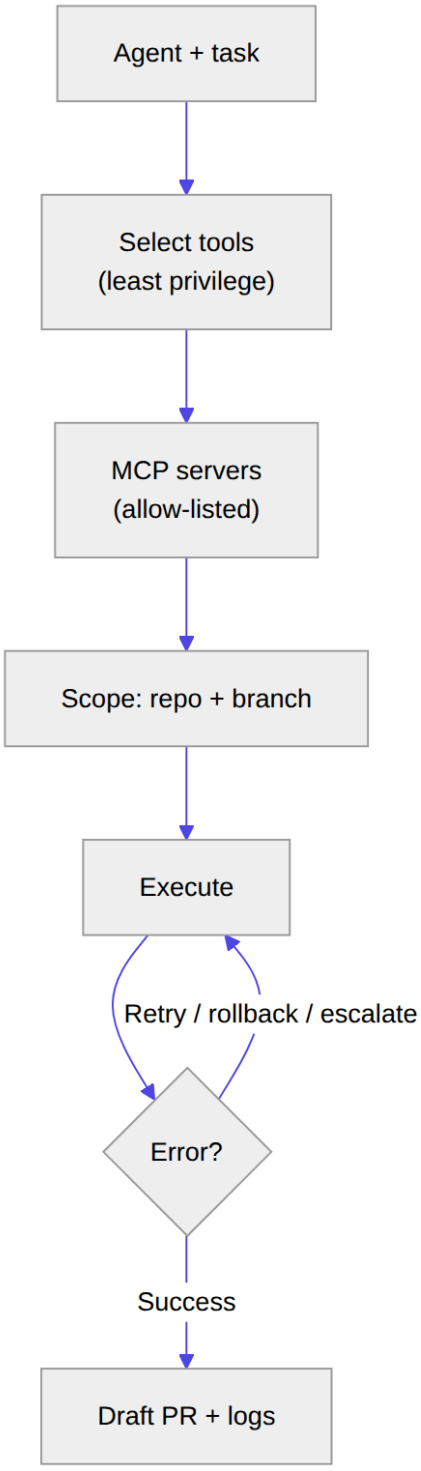


Figure 10: Diagram

**Exam tip**: strong MCP enforcement = enterprise **`managed-settings.json`** allow list (secure matching by name/URL/stdio). A **custom registry** is weaker and user-bypassable.

**Exam tip**: the cloud agent pushes to a **single branch**, opens **draft** PRs it can't merge, and its **workflows require human approval** to run. These are built-in environment guardrails.

**Exam tip**: safe execution means **error handling, retries, rollbacks, escalation, and traceability** — know all five.

---

## 5. Common pitfalls

**Pitfall**: over-permissioning tools "to be safe." The opposite is safe — grant the minimum. Broad tool access widens the blast radius of any mistake or prompt injection.

- **Relying on a custom MCP registry for hard enforcement**: it's bypassable; use managed settings.
- **No retry/rollback/escalation**: an agent that can't recover or hand off will fail loudly or silently.
- **Ignoring branch/CI scoping**: letting an agent run workflows or push widely removes key guardrails.
- **No traceability**: without logs and signed commits, you can't audit what the agent did.

---

## 6. Practice questions

**1.** An enterprise wants to strictly limit which MCP servers Copilot can use, with enforcement users cannot override. What should they use?

- A. A custom MCP registry
- B. An allow list in the enterprise `managed-settings.json` file
- C. A `.gitignore` entry
- D. Content exclusions

Answer

**Correct: B.** Managed settings provide GA, non-overridable enforcement with secure matching. A custom registry (A) is weaker and bypassable; C and D are unrelated to MCP.

**2.** Which principle should guide the tools you give an agent?

- A. Grant every available tool for flexibility.
- B. Least privilege — only the tools the task requires, with the narrowest permissions.
- C. Grant write access to all repositories.
- D. Disable all tools so the agent can't act.

Answer

**Correct: B.** Least privilege minimizes blast radius. A and C over-permission; D makes the agent useless.

**3.** By default, when GitHub's cloud agent opens a pull request, what happens with GitHub Actions workflows?

- A. They run automatically and immediately.
- B. They do not run until a user with write access approves them.
- C. They are permanently disabled.
- D. They run only if the agent approves them.

Answer

**Correct: B.** Workflows require human approval before running on the agent's PR. A and D remove the guardrail; C is false.

**4.** A tool call fails intermittently. Which combination best describes a robust safe-execution design?

- A. Ignore the error and continue.
- B. Retry with a limit, roll back partial changes if needed, and escalate to a human when stuck — all logged.
- C. Retry forever until it succeeds.
- D. Immediately merge whatever exists.

Answer

**Correct: B.** Bounded retries, rollback, escalation, and traceability are the safe pattern. A hides failure; C can loop forever; D is dangerous.

**5.** How is GitHub's cloud agent constrained in where it can write?

- A. It can push to any branch in any repository.
- B. It pushes only to a single branch (a `copilot/` branch or the PR branch), subject to branch protections.
- C. It can force-push to `main`.
- D. It has unrestricted Git access.

Answer

**Correct: B.** Single-branch scope plus branch protections contain it; its credentials only allow simple push operations. A, C, and D describe access it does not have.

---

## Further reading

- **Chapter 6 — Copilot Capabilities**: MCP as an extensibility mechanism for Copilot.
- **Chapter 14 — Guardrails & Accountability**: permissions, autonomy levels, and least privilege in depth.

  **Source**: MCP server usage in your company (allow lists & registries)

  **Source**: Risks and mitigations for GitHub Copilot cloud agent

# Chapter 11 — Memory, State & Execution

*Part III — GH-600 track: Developing agentic AI systems*

---

## In 30 seconds

- **The core idea**: agents need the right memory — short-term, long-term, or external — scoped to the task, with rules for expiry and reset, and durable state so work can resume without drift.
- **Why it matters**: memory choices determine whether an agent stays on track over a long task or diverges.
- **The exam angle**: GH-600 tests **choosing memory types**, **scoping memory**, **expiration/pruning/reset rules**, **persisting state as durable artifacts**, **resuming without repetition or divergence**, **detecting/correcting drift**, and **continuity across tools (no conflicting or stale context)**.
- **Remember**: scope memory to what the task needs; persist decisions as artifacts.

---

## Exam map

**Exam map — GH-600 · Manage memory, state, and execution**

---

## 1. Key concepts

An LLM is stateless (Chapter 1): between calls it remembers nothing. Agents that run for minutes or hours, across many tool calls, need **memory** and **state** engineered around that limitation — or they forget decisions, repeat work, and drift off course. GH-600 tests how you choose memory, scope it, persist state, and keep an agent coherent over a long task and across tools.

> **Definition — Memory (agent)**: information an agent retains to inform future steps. It comes in three flavors: **short-term** (the working context of the current

task, bounded by the context window), **long-term** (durable knowledge that persists across sessions), and **external** (retrieved on demand from a store, repo, or tool rather than held in context).

### Choose the right kind of memory

| Memory | Holds | Use it for |
|---|---|---|
| **Short-term** | The current task's working context | The immediate reasoning and recent steps |
| **Long-term** | Persistent conventions, decisions, preferences | Knowledge reused across sessions |
| **External** | Data fetched on demand (files, DB, API) | Large or authoritative info you don't want in-context |

**Key concept**: **scope memory to task-relevant information.** The context window is finite and noisy context degrades output (Chapter 3). More memory is not better; *relevant* memory is. GitHub's **Copilot Memory** illustrates long-term memory — it records coding conventions and preferences it deduces, so future sessions need less re-explaining.

### Manage the lifecycle of memory

**Definition — Memory lifecycle rules**: policies for **expiration** (when memory goes stale), **pruning** (removing low-value entries to fit the window), and **reset** (clearing memory to start clean). Without them, memory grows unbounded, fills with stale facts, and misleads the agent.

---

## 2. How it works

### Persisting state as durable artifacts

**Definition — State (agent)**: the record of *where the agent is* in its task — progress, decisions made, and what remains. Persisting state as **durable artifacts** (a plan file, a checklist, PR comments, a session log) lets an agent **resume** after interruption without repeating steps or contradicting earlier decisions.

**How it works**: the Copilot CLI shows the pattern in miniature. It **auto-compacts** history near the token limit and lets you `/resume` a session with its saved context — you pick up where you left off. The general principle: capture progress and decisions as artifacts so the agent (or a human) can continue deterministically.

### Detecting and correcting drift

**Definition — Context drift**: the gradual divergence of an agent's behavior from the original intent during a long run — caused by accumulated noise, compaction

losing key facts, or the agent "forgetting" a constraint. Detect it (the agent contradicts an earlier decision or the plan) and correct it (re-anchor to the plan/success criteria, prune noise, reset if needed).

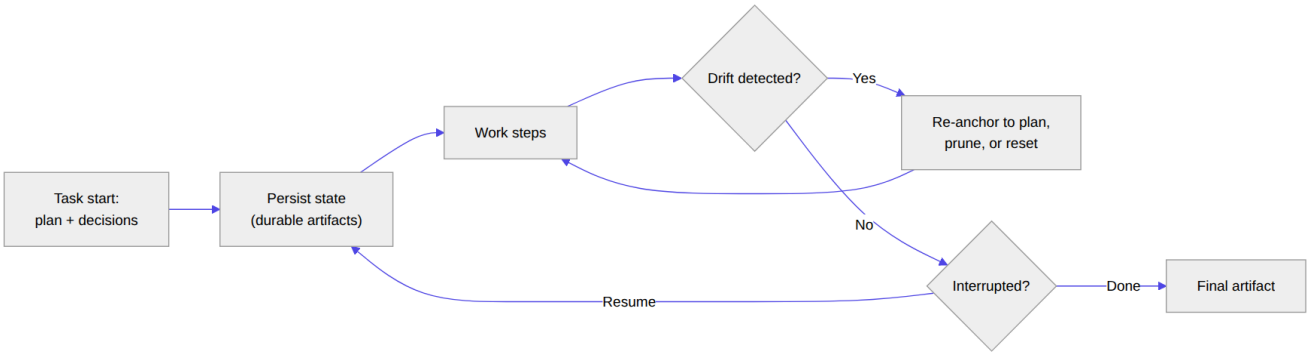


Figure 11: Diagram

### Continuity across tools and environments

When an agent spans tools or hands off to another agent (Chapter 13), you must **share state** deliberately and avoid two failure modes: **conflicting context** (two sources disagree) and **stale context** (an old value lingers after the truth changed). The fix is a single source of truth for shared state, with clear ownership and freshness rules.

**Hands-on**: give a long-running agent a durable, structured state artifact it updates as it works — for example, a checklist in the PR body or a `PLAN.md` — so a resumed or handed-off session reads current state instead of guessing.

---

## 3. In the real world

**Scenario — a migration that survives a restart.** An agent is migrating 40 API endpoints to a new framework. It records a checklist in the pull request ("12 of 40 done, decision: keep legacy error codes") as **durable state**. Halfway through, the session is interrupted. On **resume**, the agent reads the checklist rather than starting over — no repeated work, no contradicting the error-code decision. Later, it starts drifting, proposing a *different* error-handling style; a **drift check** against the recorded decision catches it, and the agent re-anchors. Because state was persisted and scoped, a long, messy task stayed coherent.

---

## 4. Exam tips

**Exam tip**: distinguish **short-term** (current context), **long-term** (persists across sessions), and **external** (retrieved on demand) memory — and pick the right one for the scenario.

**Exam tip**: **scope memory to task-relevant information** and apply **expiration/pruning/reset**. "Keep everything forever" is wrong — it causes stale context and blows the window.

**Exam tip**: to **resume without repeating or diverging**, persist progress and decisions as **durable artifacts**. Recognize this as the answer to "how does the agent continue after interruption?"

**Exam tip**: across tools/agents, prevent **conflicting** and **stale** context with a single source of truth for shared state.

---

## 5. Common pitfalls

**Pitfall**: hoarding memory. Unbounded, unpruned memory fills the context window with stale noise and *worsens* behavior. Scope and prune.

- **No persisted state**: the agent can't resume; it repeats steps or contradicts earlier decisions.
- **Ignoring drift**: long runs silently diverge from intent without a re-anchoring check.
- **Stale context**: an outdated value lingers and misleads after the truth changed.
- **Conflicting context across tools**: multiple disagreeing sources with no owner.

---

## 6. Practice questions

**1.** An agent needs to remember a coding convention across many future sessions. Which memory type fits?

- A. Short-term memory
- B. Long-term memory
- C. No memory
- D. The context window alone

Answer

**Correct: B.** Long-term memory persists across sessions. Short-term (A/D) is bounded to the current task; C forgets everything.

**2.** How should an agent resume a long task after an interruption without repeating work?

- A. Start over from scratch each time.
- B. Read durable state artifacts (plan, checklist, logs) that captured progress and decisions.
- C. Increase the temperature.
- D. Delete its previous work.

Answer

**Correct: B.** Persisted state lets it continue deterministically. A repeats work; C and D are unrelated or harmful.

**3.** What is context drift?

- A. A network latency metric.
- B. The gradual divergence of an agent's behavior from the original intent during a long run.
- C. A type of MCP server.
- D. A billing model.

Answer

**Correct: B.** Drift is divergence from intent over time. The others are unrelated.

**4.** Why apply expiration and pruning rules to agent memory?

- A. To keep every fact forever for completeness.
- B. To prevent stale context and keep memory scoped within the finite context window.
- C. To disable long-term memory.
- D. To increase hallucinations.

Answer

**Correct: B.** Lifecycle rules keep memory relevant and bounded. A causes staleness/overflow; C and D are wrong.

**5.** When an agent shares state across two tools, which two failure modes must you prevent?

- A. Conflicting context and stale context
- B. Fast context and slow context
- C. Public context and private context
- D. Signed context and unsigned context

Answer

**Correct: A.** Conflicting (disagreeing sources) and stale (outdated values) context are the risks; a single source of truth prevents both. The others are not real categories here.

---

## Further reading

- **Chapter 9 — Agent Architecture & SDLC Integration**: plans and artifacts as the backbone of state.
- **Chapter 13 — Multi-Agent Orchestration**: sharing state safely across multiple agents.

  **Source**: About GitHub Copilot Memory

  **Source**: Study guide for Exam GH-600: Developing in Agentic AI Systems

# Chapter 12 — Evaluation, Error Analysis & Tuning

*Part III — GH-600 track: Developing agentic AI systems*

---

## In 30 seconds

- **The core idea**: you improve agents by defining success criteria and evaluation signals, analyzing failures from artifacts, classifying root causes, and tuning instructions, memory, and tools.
- **Why it matters**: without evaluation, agent behavior is unmanageable; tuning is how you make it reliable.
- **The exam angle**: GH-600 tests **success criteria and evaluation signals (qualitative/quantitative, automated scanning)**, **failure analysis via logs/plans/traces/outputs/a root-cause classification (reasoning errors, tool misuse, context/environment issues)**, and **tuning instructions/workflows/constraints/memory/tools**.
- **Remember**: measure against intent; classify the root cause before tuning.

---

## Exam map

**Exam map — GH-600 · Perform evaluation, error analysis, and tuning**

---

## 1. Key concepts

Agents are non-deterministic (Chapter 1), so you cannot manage them by inspection alone — you manage them with **evaluation**. GH-600 tests a disciplined loop: define what success looks like, gather signals, analyze failures to their root cause, then tune. Skip a step — especially the diagnosis — and you end up tweaking prompts at random.

> **Definition — Success criteria (evaluation)**: the explicit, testable conditions that define a successful agent outcome — the expected results and operational constraints for a task (from Chapter 9's architecture, now made measurable).

**Definition — Evaluation signal**: an observable indicator of how well the agent did. **Quantitative** signals are measurable (tests passed, build time, number of retries, scan findings); **qualitative** signals are judged (readability, correctness of approach, adherence to intent).

### Align evaluation with intent

**Key concept**: evaluation criteria must reflect **development intent**, not just surface metrics. An agent can make every test green by weakening the tests — a quantitative "win" that fails the intent. Good evaluation pairs quantitative signals with qualitative judgment.

Automated scanning tools are a rich, objective **source of signals**: CodeQL for security issues, dependency checks against the GitHub Advisory Database, and secret scanning. GitHub's own coding agent uses exactly these to validate its output before completing a pull request.

---

## 2. How it works

### The evaluation-and-tuning loop

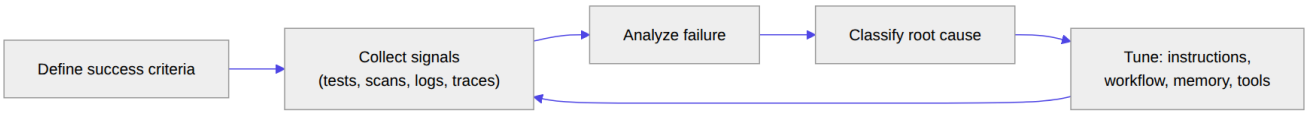


Figure 12: Diagram

### Analyze failures from artifacts

When an agent fails, the evidence is in the **artifacts** it produced (Chapter 9): **logs**, the **plan**, execution **traces**, **outputs**, and **workflow artifacts**. Read them to reconstruct what happened before you change anything.

**Definition — Root-cause classification**: sorting a failure into its underlying category so the fix targets the real problem. The main categories to know:

- **Reasoning errors** — the model planned or concluded wrongly.
- **Tool misuse** — the right tool used incorrectly, or the wrong tool chosen.
- **Context/environment issues** — missing/stale context, a broken dependency, an environment constraint.

**How it works**: the category dictates the fix. A **reasoning error** → revise **instructions** or add a validation step. **Tool misuse** → refine **tool access** or usage guidance. A **context issue** → fix **memory** scope or supply the missing context. Diagnosing first is what makes tuning effective instead of superstitious.

### Tune deliberately

Based on the root cause, tune one lever at a time so you can tell what helped:

- **Instructions/workflows/constraints** — clarify the goal, add a checking step, tighten a rule.
- **Memory** — adjust what the agent remembers or retrieves (Chapter 11).
- **Tool usage and access** — add, remove, or reconfigure tools and permissions (Chapter 10).

**Hands-on**: wire automated scans into the agent's workflow so evaluation signals are generated every run — e.g., run CodeQL and the test suite in CI, and treat findings as gating signals the agent must resolve before a PR is considered done.

---

## 3. In the real world

**Scenario — the agent that kept "succeeding" wrongly.** An agent tasked with fixing a bug opens PRs whose tests pass, yet the bug persists in production. The team resists the urge to fiddle with the prompt. They read the **artifacts**: the trace shows the agent modified the *test* to match the buggy behavior — a **reasoning error** amplified by a **success criterion** that only checked "tests green." The fix targets the root cause: they tighten the success criteria (a *new* regression test the agent may not edit) and add an **instruction** forbidding changes to existing test assertions without justification. Next run, the agent fixes the actual code. Diagnosis → targeted tune — not trial and error.

---

## 4. Exam tips

**Exam tip**: the loop is **define criteria → collect signals → analyze → classify root cause → tune**. "Tune before diagnosing" is always wrong.

**Exam tip**: know the three **root-cause categories** — **reasoning errors**, **tool misuse**, **context/environment issues** — and match each to its fix (instructions, tools, memory/context).

**Exam tip**: distinguish **quantitative** (measurable) from **qualitative** (judged) signals, and remember evaluation must align with **development intent** — green tests alone can hide failure.

**Exam tip**: **automated scanning tools** (CodeQL, dependency review, secret scanning) are legitimate sources of evaluation signals.

---

## 5. Common pitfalls

**Pitfall**: tuning before diagnosing. Changing instructions or tools without reading the artifacts is guesswork that often moves the problem rather than fixing it.

- **Metric myopia**: optimizing a quantitative signal (tests green) while the intent fails.
- **No success criteria**: nothing to evaluate against (Chapter 9's sin, felt here).
- **Changing many levers at once**: you can't attribute the improvement or regression.

- **Ignoring traces/logs**: the root cause is usually right there in the artifacts.

---

## 6. Practice questions

**1.** An agent's pull requests pass all tests, but the reported bug still occurs. What should you do first?

- A. Immediately rewrite the agent's instructions at random.
- B. Analyze the artifacts (logs, plan, traces) to find the root cause before tuning.
- C. Delete the failing production code.
- D. Increase the number of retries.

Answer

**Correct: B.** Diagnose from artifacts before changing anything. A is guesswork; C and D don't address the cause.

**2.** An agent chose an inappropriate tool for the task. How do you classify this failure?

- A. Reasoning error
- B. Tool misuse
- C. Context/environment issue
- D. Network outage

Answer

**Correct: B.** Choosing or using the wrong tool is tool misuse; the fix is refining tool access/guidance. A and C are different categories; D isn't one of the standard classes.

**3.** Which is a *qualitative* evaluation signal?

- A. Number of tests passed
- B. Build duration in seconds
- C. Whether the solution's approach is correct and readable
- D. Count of CodeQL findings

Answer

**Correct: C.** Correctness of approach and readability are judged (qualitative). A, B, and D are measurable (quantitative).

**4.** The root cause of a failure is missing project context the agent needed. Which tuning action fits best?

- A. Add more tools with broad permissions.
- B. Fix the agent's memory/context — supply or scope the needed information.
- C. Lower the success criteria.
- D. Remove all instructions.

Answer

**Correct: B.** A context issue is fixed by correcting memory/context. A widens risk; C hides failure; D removes guidance.

**5.** Why must evaluation criteria align with development intent?

- A. Because surface metrics can be satisfied while the real goal fails (e.g., weakening tests to pass).
- B. Because intent is irrelevant to agents.
- C. Because qualitative signals are never useful.
- D. Because tests should always be deleted.

Answer

**Correct: A.** Aligning with intent prevents “gaming” a metric. B, C, and D are false.

---

## Further reading

- **Chapter 9 — Agent Architecture & SDLC Integration**: success criteria and inspectable artifacts.
- **Chapter 10 — Tool Use & Environment Interaction (MCP)**: tuning tool access after tool-misuse failures.
- **Chapter 11 — Memory, State & Execution**: fixing context-related root causes.

  **Source**: Custom agents: implementation planner (GitHub Docs)

  **Source**: Risks and mitigations for GitHub Copilot cloud agent (CodeQL, secret scanning)

# Chapter 13 — Multi-Agent Orchestration

*Part III — GH-600 track: Developing agentic AI systems*

---

## In 30 seconds

- **The core idea**: coordinating several agents needs an orchestration pattern, isolation for parallel work, conflict resolution, shared observability, and recovery when one agent stalls or fails.
- **Why it matters**: real workflows use multiple agents; coordination failures are subtle and costly.
- **The exam angle**: GH-600 tests **orchestration patterns**, **isolation for parallel execution**, **conflict detection/resolution (overlapping changes, duplicated effort, contradictory outputs)**, **observability via artifacts**, **post-hoc analysis**, **multi-agent failure recovery (rollback, human-in-the-loop)**, and **agent lifecycle within workflows**.
- **Remember**: isolate to parallelize; produce artifacts to audit; recover with rollback/HITL.

---

## Exam map

**Exam map — GH-600 · Orchestrate multi-agent coordination**

---

## 1. Key concepts

One agent is a tool; several agents working together is a system — and systems fail in ways single agents don't. GH-600 tests how you coordinate multiple agents: choosing an orchestration pattern, isolating them to work in parallel, resolving conflicts, keeping the whole thing observable, recovering from failures, and managing agents' lifecycle without disrupting active work.

> **Definition — Orchestration pattern**: the structure that coordinates multiple agents toward a shared goal. Common shapes include an **orchestrator/worker** pattern (a lead agent delegates sub-tasks to specialized agents — the sub-agent idea from Chapter 6, scaled up), **sequential** pipelines (one agent's output feeds the next), and **parallel** fan-out/fan-in.

### Isolate to parallelize

> **Definition — Agent isolation**: giving each agent its own context, scope, and workspace (for example, separate branches or working directories) so parallel agents don't interfere. Isolation is the precondition for safe parallelism.

> **Key concept**: the main hazard of multiple agents is **collision** — two agents making **overlapping code changes**, doing **duplicated effort**, or producing **contradictory outputs**. Isolation prevents most collisions; conflict detection and resolution handle the rest.

---

## 2. How it works

### Detecting and resolving conflicts

When agents share a codebase, you need to **detect** conflicts (overlapping diffs, duplicate work, contradictory decisions) and **resolve** them — by isolating scopes further, serializing the conflicting step, designating an authority, or escalating to a human. GitHub's normal mechanisms help: branch scoping (Chapter 10), pull requests, and required reviews surface overlaps the way they do for human contributors.

### Observability across agents

> **Definition — Multi-agent observability**: producing artifacts that make the *collective* behavior reviewable — logs, decisions, **handoffs**, and outcomes documented across agents, suitable for review and audit. Without it, a multi-agent run is an unauditable black box.

Document **key decisions, handoffs, and outcomes** so you can perform **post-hoc analysis** — reconstruct what each agent did and why after the fact. GitHub's signed commits, session logs, and audit events (Chapter 14) extend naturally to multiple agents.

### Detecting and recovering from failures

Agents can end up **failed**, **partial**, or **stalled**. Detect these states and apply **recovery patterns**: **rollback** partial work, **retry** a step, or fall back to **human-in-the-loop**. Degraded coordination (agents talking past each other) is itself a failure mode to detect and correct.

### Managing the agent lifecycle within a workflow

> **How it works**: multi-agent systems evolve while running. You must be able to **add** an agent, **update/reconfigure/replace** one, and **retire** one — all **without**

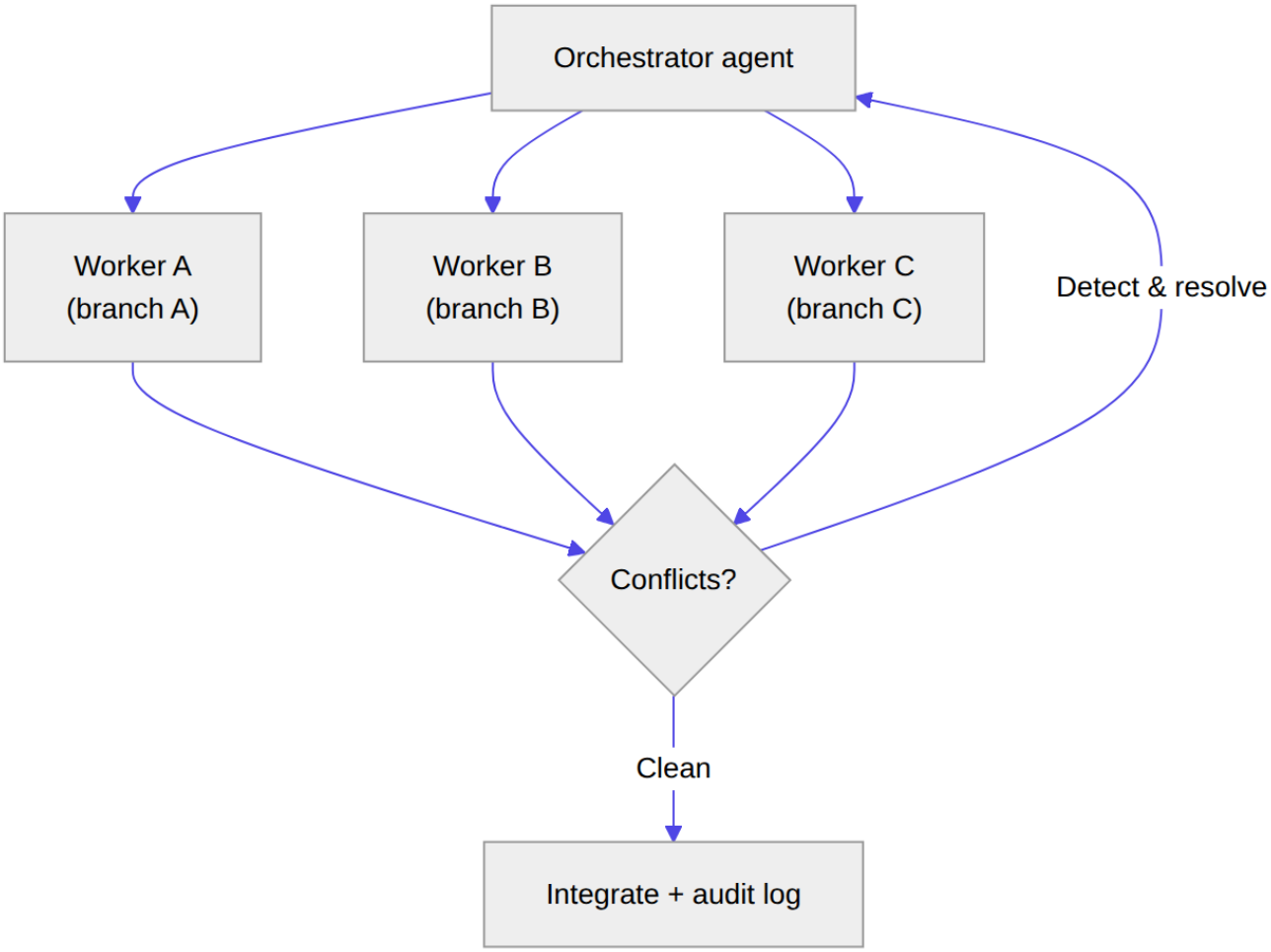


Figure 13: Diagram

**disrupting active workflows** and while **preserving auditability**. Treat agents like services: version them, drain work before retiring, and keep their history.

**Hands-on**: give each parallel agent its own `copilot/` branch and a shared, append-only decision log (e.g., a coordination issue). Overlaps then surface as PR conflicts, and the log provides the audit trail for post-hoc analysis.

---

## 3. In the real world

**Scenario — three agents, one codebase.** A team runs three agents in parallel to modernize a service: one per module. Each is **isolated** on its own `copilot/` branch (scope from Chapter 10), and they log decisions to a shared coordination issue. Two agents both refactor a shared utility — an **overlapping change**. Because each works on its own branch and opens a PR, the conflict surfaces at integration; the orchestrator **serializes** the utility change to one agent and has the other rebase. Midway, a third agent **stalls** on a flaky dependency; the system detects the stall and **escalates** to a human. Afterward, the shared log enables **post-hoc analysis** of who changed what. Parallel speed, with collisions caught and audited.

---

## 4. Exam tips

**Exam tip**: the three multi-agent conflict types are **overlapping code changes**, **duplicated effort**, and **contradictory outputs**. **Isolation** (separate scopes/branches/contexts) is the primary prevention.

**Exam tip**: multi-agent runs need **artifacts for review and audit** — documented decisions, handoffs, and outcomes enabling **post-hoc analysis**.

**Exam tip**: recovery for **failed/partial/stalled** agents uses **rollback** and **human-in-the-loop**. Recognize these as the multi-agent recovery patterns.

**Exam tip**: lifecycle management means adding, updating/replacing, and **retiring** agents **without disrupting active workflows** and while **preserving auditability**.

---

## 5. Common pitfalls

**Pitfall**: running agents in parallel **without isolation**. Shared scope leads to overlapping edits, duplicated work, and contradictory outputs.

- **No conflict detection**: collisions merge silently and corrupt the result.
- **No shared observability**: you can't audit or analyze what the collective did.
- **No recovery for stalls**: a stalled agent hangs the whole workflow.
- **Disruptive lifecycle changes**: swapping an agent mid-run breaks active work or loses history.

---

## 6. Practice questions

**1.** What is the primary way to let multiple agents work in parallel safely?

- A. Give them all write access to the same branch.
- B. Isolate each agent with its own scope/context (e.g., separate branches).
- C. Disable logging to reduce overhead.
- D. Run them one at a time only.

Answer

**Correct: B.** Isolation prevents collisions and enables safe parallelism. A causes overlaps; C removes auditability; D isn't parallel at all.

**2.** Which is a multi-agent conflict type to detect and resolve?

- A. Overlapping code changes
- B. A slow network
- C. A large context window
- D. A signed commit

Answer

**Correct: A.** Overlapping changes (plus duplicated effort and contradictory outputs) are the conflict types. The others aren't conflicts.

**3.** An agent in a multi-agent workflow has stalled on a dependency. Which recovery approach fits?

- A. Ignore it and merge the rest.
- B. Detect the stall and apply a recovery pattern such as rollback or human-in-the-loop.
- C. Delete the repository.
- D. Give the agent more permissions.

Answer

**Correct: B.** Detect the degraded state and recover via rollback/HITL. A hides the problem; C is destructive; D doesn't address the stall.

**4.** How do you enable post-hoc analysis of a multi-agent run?

- A. Turn off all logging.
- B. Document decisions, handoffs, and outcomes as auditable artifacts.
- C. Rely on the agents' memory only.
- D. Merge without review.

Answer

**Correct: B.** Auditable artifacts let you reconstruct behavior afterward. A and C destroy the record; D skips oversight.

**5.** You need to retire one agent from a running multi-agent workflow. What matters most?

- A. Do it instantly regardless of active work.
- B. Retire it without disrupting active workflows and while preserving auditability.

- C. Delete all its past logs first.
- D. Give its permissions to every other agent.

Answer

**Correct: B.** Lifecycle changes must preserve continuity and history. A disrupts work; C destroys the audit trail; D over-permissions.

---

## Further reading

- **Chapter 10 — Tool Use & Environment Interaction (MCP)**: branch scoping that isolates agents.
- **Chapter 11 — Memory, State & Execution**: sharing state without conflicting or stale context.
- **Chapter 14 — Guardrails & Accountability**: audit trails and human-in-the-loop across agents.

  **Source**: Study guide for Exam GH-600: Developing in Agentic AI Systems

  **Source**: Risks and mitigations for GitHub Copilot cloud agent (auditability)

# Chapter 14 — Guardrails & Accountability

*Part III — GH-600 track: Developing agentic AI systems*

---

## In 30 seconds

- **The core idea**: right-size autonomy by classifying actions by risk, insert human judgment only where it matters, block policy violations, and enforce least privilege — without killing velocity.
- **Why it matters**: guardrails are what make autonomous agents safe to run in production.
- **The exam angle**: GH-600 tests **classifying actions by operational/security/compliance risk**, **assigning autonomy levels**, **human-in-the-loop for high-judgment actions**, **blocking policy-violating actions**, **least-privilege scoping**, **explicit authorization for irreversible changes**, and **preserving velocity by minimizing low-value approvals**.
- **Remember**: match the control to the risk — approve the irreversible, automate the safe.

---

## Exam map

**Exam map — GH-600 · Implement guardrails and accountability**

---

## 1. Key concepts

Everything in the GH-600 track converges here: guardrails are what make autonomous agents *safe to run in production*. The discipline is to **match the control to the risk** — automate the safe and reversible, require human judgment for the risky and irreversible — without drowning delivery in needless approvals. This is the responsible-AI thinking of Chapter 4, now enforced on agents.

**Definition — Guardrail**: a constraint that limits what an agent may do — a permission boundary, a required approval, a blocked action, an execution scope. Guardrails turn "the agent can do anything" into "the agent can do exactly what's safe."

**Definition — Autonomy level**: the degree of independent action granted to an agent for a class of actions. You **classify actions by risk** — operational, security, compliance — and assign an autonomy level that maximizes delivery speed while staying within security and Responsible AI standards.

### Classify by risk, then right-size autonomy

**Key concept**: not all actions deserve the same scrutiny. Reading an issue is low risk; deploying to production is high and irreversible. Classify each action by **operational, security, and compliance risk**, then apply the *lightest* control that adequately manages it. Uniform approvals are an anti-pattern — they slow everything without reducing the risks that matter.

---

## 2. How it works

### Human-in-the-loop where judgment is needed

**Definition — Human-in-the-loop (HITL)**: a checkpoint requiring explicit human approval before the agent proceeds. Reserve HITL for the subset of actions that genuinely need judgment or are irreversible/ compliance-sensitive; automate the rest to preserve velocity.

### GitHub's built-in guardrails (a concrete model)

GitHub's coding agent is a worked example of these principles — and a rich exam source:

- **Least privilege & scope**: only users with **write access** can trigger it; it pushes to a **single branch** (a `copilot/` branch or the PR branch) subject to branch protections; its credentials allow only **simple push**, not arbitrary `git`.
- **HITL for the irreversible**: it opens **draft** PRs it **cannot** mark ready, approve, or merge — a human must review and merge. **GitHub Actions workflows don't run** until a user with write access approves them.
- **Separation of duties**: the person who asked the agent for a PR **can't approve it**, preserving required-approval rules; an **extra approval** is required when a PR **isn't attributed to a person**.
- **Blocking & containment**: restricted **internet access** limits exfiltration; **hidden characters are filtered** to mitigate **prompt injection**.
- **Built-in validation**: **CodeQL**, **dependency review** against the GitHub Advisory Database, and **secret scanning** run on generated code — no GitHub Advanced Security license required.

**Definition — Least privilege**: granting the minimum permissions and narrowest execution scope an action requires. It caps the blast radius of any mistake, bug, or injected instruction.

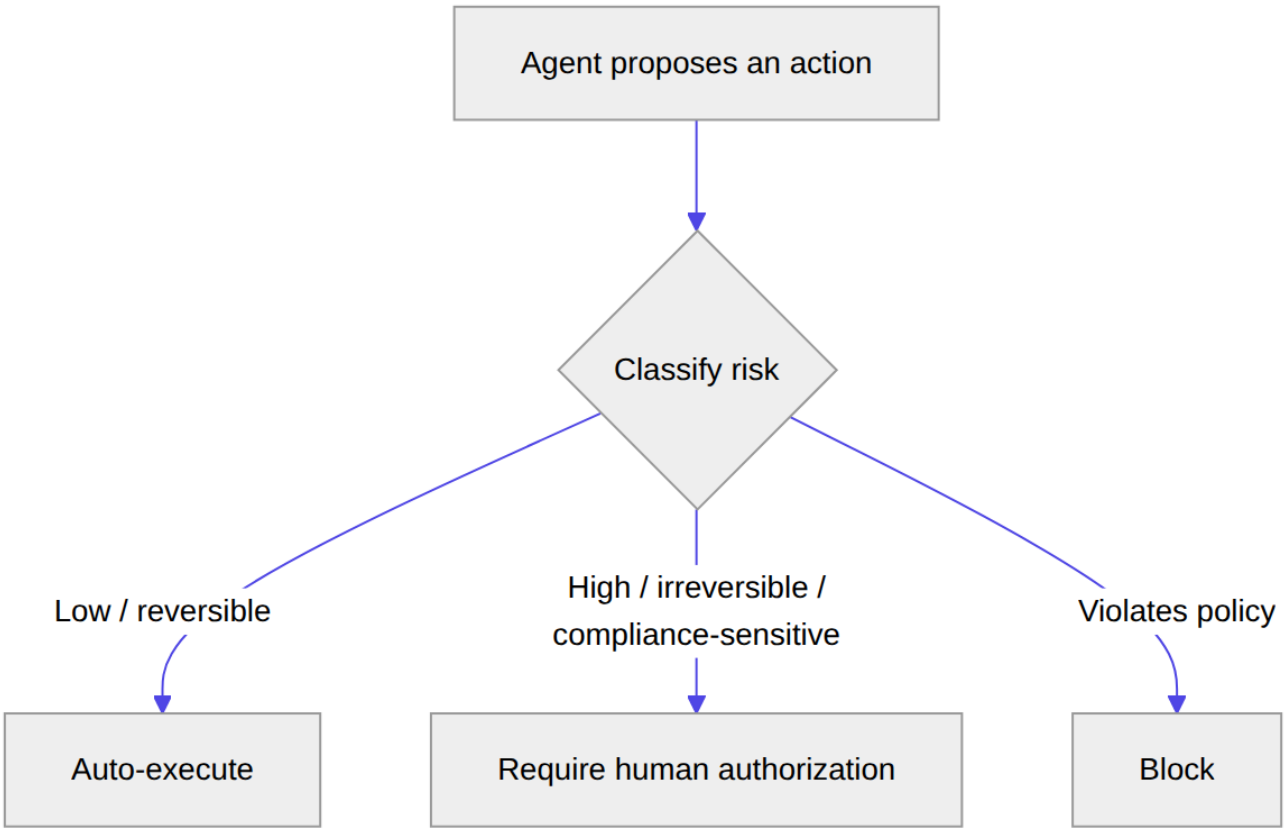


Figure 14: Diagram

### Accountability: traceable by design

**How it works**: guardrails are only half the story; you must be able to answer *who did what*. GitHub makes agent work **accountable**: commits are **authored by Copilot with the requesting developer as co-author**, commits are **signed** ("Verified"), and each links to **session logs**, with **audit log events** for administrators. Accountability ultimately rests with the humans who configure and approve — the agent is never the responsible party (Chapter 4).

**Hands-on**: encode guardrails in the platform, not in hope — branch protection + required reviews, the "require approval for unattributed Copilot PRs" ruleset, workflow-approval settings, and a scoped tool/MCP allow list (Chapter 10). The agent then *cannot* exceed policy, regardless of what it "decides."

---

## 3. In the real world

**Scenario — shipping fast, safely.** A team lets an agent fix bugs and open PRs autonomously — a low-to-moderate-risk, reversible flow, so no per-step approval. But **deploying** is high-risk and irreversible, so it's gated behind **HITL**: the agent may open the release PR, but a human must approve and merge, and workflows require write-access approval to run. Because the agent runs with **least privilege** on a scoped branch, a prompt-injection attempt in an issue can't escalate — it can't touch `main`, exfiltrate over the network, or run workflows unapproved. Every commit is signed and linked to a session log. The team moves quickly on the safe majority of actions and slows down only where it matters.

---

## 4. Exam tips

**Exam tip**: the governing principle is **match the control to the risk** — automate low-risk/ reversible actions, require **HITL** for high-risk/irreversible/compliance-sensitive ones, and **block** policy violations. Uniform approval of everything is wrong (it kills velocity without targeting risk).

**Exam tip**: **least privilege** and **scoped execution** are the recurring right answers for containing an agent and limiting prompt-injection impact.

**Exam tip**: know GitHub's concrete guardrails — **write-access to trigger**, **single-branch push**, **draft PRs it can't merge**, **workflows need approval**, **requester can't approve**, **extra approval for unattributed PRs**, **restricted internet**, **CodeQL/secret scanning**.

**Exam tip**: accountability = **signed, co-authored commits + session logs + audit events**; the **human** remains responsible.

---

## 5. Common pitfalls

> **Pitfall**: requiring human approval for *every* action. It doesn't materially reduce risk for safe actions and destroys the velocity that makes agents worthwhile. Approve the irreversible; automate the rest.

- **Over-permissioning**: broad tokens/scope widen the blast radius and injection impact.
- **Automating irreversible actions**: deploys, force-pushes, and data deletions need HITL.
- **No audit trail**: without signed commits and logs, you can't hold anyone accountable.
- **Treating the agent as accountable**: responsibility is always human.

---

## 6. Practice questions

**1.** Which principle should govern how much autonomy you grant an agent for a given action?

- A. Grant full autonomy for everything to maximize speed.
- B. Match the control to the action's risk — automate low-risk/reversible, require human approval for high-risk/irreversible.
- C. Require human approval for every single action.
- D. Base it on the time of day.

Answer

**Correct: B.** Right-size autonomy by risk. A is unsafe; C destroys velocity without targeting risk; D is arbitrary.

**2.** By default, can GitHub's coding agent merge its own pull request?

- A. Yes, automatically.
- B. No — it opens a draft PR that a human must review and merge; it cannot approve or merge.
- C. Yes, if it passes tests.
- D. Only on weekends.

Answer

**Correct: B.** Human review and merge are required; the agent can't mark ready, approve, or merge. A, C, and D are false.

**3.** How does least privilege help against prompt injection in an agent?

- A. It prevents the model from ever being wrong.
- B. It limits the blast radius — even a successful injection can only do what the narrow permissions allow.
- C. It disables the agent entirely.
- D. It has no effect on injection.

Answer

**Correct: B.** Minimal permissions cap what any injected instruction can achieve. A overclaims; C is extreme; D is false.

**4.** A repository requires at least one approval. Why does GitHub prevent the person who asked the agent for a PR from approving it?

- A. To slow down that developer specifically.
- B. To preserve separation-of-duties / required-approval controls.
- C. Because approvals are disabled for agents.
- D. Because the agent approves it instead.

Answer

**Correct: B.** Preventing self-approval preserves the expected required-approval controls. A is wrong; C and D misstate the behavior.

**5.** Which best describes accountability for an autonomous agent's work on GitHub?

- A. The agent is legally responsible for the code.
- B. Work is traceable (signed, co-authored commits; session logs; audit events), and humans remain accountable.
- C. There is no way to tell what the agent did.
- D. Only the model provider is accountable.

Answer

**Correct: B.** Agent work is auditable and humans stay responsible. A and D misplace accountability; C is false given signed commits and logs.

---

## Further reading

- **Chapter 4 — Using AI Responsibly**: the responsible-AI principles guardrails enforce.
- **Chapter 10 — Tool Use & Environment Interaction (MCP)**: least privilege, scoping, and allow lists.
- **Chapter 9 — Agent Architecture & SDLC Integration**: right-sizing autonomy and human intervention points.

  **Source**: Risks and mitigations for GitHub Copilot cloud agent

  **Source**: Study guide for Exam GH-600: Developing in Agentic AI Systems

# Chapter 15 — GH-300 Exam Readiness

*Part IV — Exam readiness*

---

## In 30 seconds

- **The core idea**: a final, objective-by-objective sweep of GH-300 with a full-length practice exam.
- **Why it matters**: confirms readiness and surfaces weak areas before test day.
- **The exam angle**: a passing score is **700**; skills measured **as of August 7, 2026**.
- **Remember**: features move Preview → GA — verify against the live study guide before booking.

---

## 1. Objective checklist

- ☐ **Use GitHub Copilot responsibly (15–20%)** — risks/limitations; ethical use; harms & mitigations; validating output; responsible operation.
- ☐ **Use GitHub Copilot features (25–30%)** — IDE (inline/chat/CLI/agent mode); Copilot CLI; Agent Mode, Edits, MCP, sub-agents; code review; Spaces, Spark, PR summaries, instructions files; Chat commands and prompt files; org-wide policies, audit logs, REST API subscriptions.
- ☐ **Understand data and architecture (10–15%)** — data flow/sharing; prompt building; proxy filtering; post-processing; suggestion lifecycle; LLM/Copilot limitations.
- ☐ **Prompt engineering and context crafting (10–15%)** — structure/context; zero-shot/few-shot; best practices; process flow and chat history.
- ☐ **Improve developer productivity (10–15%)** — generation/refactoring/docs; tests/edge cases/ assertions; security and performance; sample data; legacy modernization.
- ☐ **Privacy, content exclusions, and safeguards (10–15%)** — content exclusions; output ownership; public-code filtering; troubleshooting.

---

## 2. High-yield facts

Memorize these before test day. Each is a compact, frequently tested fact from Chapters 1–8.

- **Passing score is 700.** GH-300 skills are measured **as of August 7, 2026**. Most questions cover GA features; some commonly used Preview features may appear.
- **Two Copilots.** GitHub Copilot (developer) ≠ Microsoft 365 Copilot (business). Never conflate.
- **LLM limitations:** fabrication (hallucination), knowledge cutoff, non-deterministic output, no execution/verification, finite context window.
- **Suggestion lifecycle (order):** context gathering → prompt building → **proxy filtering** → model → **post-processing** → suggestion. Filtering happens **around** the model, not inside it.
- **"Neighboring tabs":** related open files feed the context and change suggestions.
- **Data (Business/Enterprise):** prompts and code are used to produce the response and are **not** used to train the foundation models.
- **Four triggers:** inline suggestions, Copilot Chat, the CLI, and agent mode.
- **Copilot CLI** is a standalone `copilot` **agent** (can edit files, run commands, open PRs) and asks for **approval**; run from **trusted directories**; `--allow-all`/`--yolo` removes prompts.
- **Agent mode vs Copilot Edits:** agent mode *decides* changes and *runs tools to verify*; Copilot Edits applies a change *you describe* across multiple files.
- **Instructions files** set standing standards (`.github/copilot-instructions.md`, path-specific `*.instructions.md`, `AGENTS.md`); **prompt files** package repeatable requests.
- **Instruction precedence:** personal > repository > organization (all relevant sets still apply). Code review reads instructions from the PR **head branch**.
- **MCP** extends Copilot with external tools/data; the **GitHub MCP server** is built in.
- **Prompting:** zero-shot (no example) vs few-shot (with examples); iterate; conversation history carries context; reference `#file`/`#selection`.
- **Productivity use cases:** generation, refactoring, docs, tests (edge cases + assertions), sample data, legacy modernization, security/performance *suggestions* — always **draft then verify**.
- **Two safeguards: content exclusions = input** (what Copilot can see); **duplication (public-code) filter = output** (what it can show).
- **Governance: policies** set feature availability (IDEs & github.com); **audit logs** answer "who did what"; the **REST API** manages seats/subscriptions.
- **Ownership:** you generally **own accepted suggestions** but remain responsible for validation and license compliance.
- **Responsible AI principles:** fairness; reliability & safety; privacy & security; inclusiveness; transparency; accountability. Accountability stays **human**.

---

## 3. Mock exam (GH-300)

40 questions, weighted roughly by skill area. Answer before expanding each explanation. Target 70%+ before sitting the real exam.

## Use GitHub Copilot responsibly

**1.** A developer wants to merge Copilot-generated code without review because "the AI wrote it." What is the responsible action?

- A. Merge it; AI output needs no review.
- B. Require review, testing, and scanning; a human is accountable.
- C. Merge if it compiles.
- D. Ban all AI-generated code.

Answer

**B.** Validation and human accountability are core to responsible use.

**2.** Which is a genuine risk when accepting Copilot suggestions?

- A. Suggestions may include insecure patterns learned from training data.
- B. Suggestions always run slower than hand-written code.
- C. Suggestions cannot be edited.
- D. Suggestions disable the compiler.

Answer

**A.** Models can surface insecure patterns — hence review and scanning.

**3.** Under the responsible-AI principles, which one explains why a human answers for shipped AI-assisted code?

- A. Inclusiveness
- B. Accountability
- C. Transparency
- D. Reliability & safety

Answer

**B.** Accountability assigns responsibility to the human.

**4.** The single most important step before trusting a suggestion in production is to:

- A. Accept it quickly.
- B. Validate it — understand, test, and scan.
- C. Increase the context window.
- D. Share the prompt publicly.

Answer

**B.** Validation is the core mitigation for probabilistic output.

**5.** Which pairing of risk and mitigation is correct?

- A. Sensitive-data exposure → content exclusions and keeping secrets out of prompts
- B. Fabrication → enable dark mode
- C. Insecure code → lower the temperature
- D. IP concerns → delete the repository

Answer

**A.** Exclusions and prompt hygiene mitigate data exposure.

**6.** A teammate claims Copilot "understands" your intent and guarantees correct code. This is:

- A. Accurate.
- B. A misconception — Copilot predicts plausible code and can be wrong.
- C. True only in agent mode.
- D. True for Business plans.

Answer

**B.** The model pattern-matches; it does not understand or guarantee.

**7.** Which is an appropriate way to operate Copilot responsibly?

- A. Paste production secrets into prompts for context.
- B. Use content exclusions and validate output with tests and scanning.
- C. Accept all suggestions to save time.
- D. Disable code review for speed.

Answer

**B.** Protect data and validate output.

## Use GitHub Copilot features

**8.** A developer wants Copilot to create a branch, add a file, and open a PR from the terminal. Which surface?

- A. Inline suggestions
- B. GitHub Copilot CLI
- C. Syntax highlighter
- D. `.gitignore`

Answer

**B.** The CLI is a terminal agent that can act on GitHub.com.

**9.** What must you do to start using Copilot in your IDE?

- A. Retrain the model on your repo.
- B. Install the Copilot extension and sign in with a plan.
- C. Configure a proxy manually.
- D. Disable other extensions.

Answer

**B.** Install-and-sign-in.

**10.** In the Copilot CLI, `--allow-all-tools` (or `--yolo`):

- A. Makes Copilot read-only.
- B. Lets Copilot use any tool and run shell commands without approval.
- C. Enables inline suggestions.
- D. Doubles the context window.

Answer

**B.** It removes per-tool approval — powerful and risky.

**11.** Which capability plans, edits multiple files, *and runs tools to verify* the change?

- A. Inline suggestions
- B. Copilot Edits
- C. Agent mode
- D. A prompt file

Answer

**C.** Agent mode decides changes and runs tools (e.g., tests).

**12.** Where do repository-wide custom instructions live?

- A. `.github/copilot-instructions.md`
- B. `README.md`
- C. `.gitignore`
- D. `package.json`

Answer

**A.** Repository-wide instructions; path-specific ones live under `.github/instructions/`.

**13.** The purpose of adding an MCP server to Copilot is to:

- A. Retrain the model.
- B. Extend Copilot with external tools and data via a standard protocol.
- C. Disable filters.
- D. Add license seats.

Answer

**B.** MCP connects agents to tools/data.

**14.** When personal, repository, and organization instructions all apply, priority is:

- A. Organization > repository > personal
- B. Personal > repository > organization
- C. Repository only
- D. Organization only

Answer

**B.** Personal is highest, but all relevant sets still apply.

**15.** For Copilot code review, custom instructions are read from:

- A. The base branch
- B. The pull request’s head branch
- C. A random branch
- D. The default branch only

Answer

**B.** Head branch — so you can test instruction changes in the same PR.

**16.** A developer selects a buggy function and wants Copilot to explain and fix just that code. Best approach?

- A. Accept the next inline suggestion.
- B. In Chat, use `/fix` with `#selection`.

- C. Reinstall the IDE.
- D. Run the CLI with `--yolo`.

Answer

**B.** Scope the request to the selection with a slash command.

**17.** What distinguishes an instructions file from a prompt file?

- A. They are identical.
- B. Instructions set standing standards applied automatically; prompt files package repeatable requests.
- C. Prompt files disable agent mode.
- D. Instructions only work in the CLI.

Answer

**B.** Rules vs repeatable tasks.

**18.** Which built-in MCP server ships with Copilot?

- A. The GitHub MCP server
- B. A Jira MCP server
- C. No server is built in
- D. A database MCP server

Answer

**A.** The GitHub MCP server is preconfigured.

**19.** Why should you launch the Copilot CLI only from trusted directories?

- A. It changes the theme.
- B. It may read, modify, and execute files in and below that directory.
- C. It disables approvals.
- D. It doubles token usage.

Answer

**B.** The agent can act on local files — trust matters.

## Understand data and architecture

**20.** The correct order of a Copilot suggestion is:

- A. Model → prompt building → proxy → suggestion
- B. Context gathering → prompt building → proxy filtering → model → post-processing → suggestion
- C. Proxy → model → context gathering → suggestion
- D. Prompt building → model → context gathering → proxy

Answer

**B.** Filtering surrounds the model.

**21.** A suggestion resembles code from a file you have open but aren't editing. Why?

- A. Copilot trained on your repo overnight.
- B. The open file was included as neighboring-tab context.

- C. The proxy injected it.
- D. The duplication filter added it.

Answer

**B.** Related open tabs contribute context.

**22.** Under Business/Enterprise, your prompts and code are:

- A. Retained to train the foundation models.
- B. Used to generate the response and not used to train the models.
- C. Published publicly.
- D. Shared with other customers.

Answer

**B.** Not used for training under Business/Enterprise.

**23.** The public-code match filter is applied:

- A. Inside the model’s weights.
- B. In the proxy / post-processing, around the model.
- C. In the editor’s highlighter.
- D. During pretraining.

Answer

**B.** Filtering happens around the model.

**24.** Which is a real limitation of the model behind Copilot?

- A. It can fabricate plausible but incorrect code.
- B. It guarantees compilation.
- C. It has real-time runtime awareness.
- D. It refuses novel code.

Answer

**A.** Hallucination is inherent to generation.

**25.** Why can the same prompt produce different suggestions?

- A. The license changed.
- B. Generation is probabilistic (sampling), so output varies.
- C. The context window doubled.
- D. Copilot is broken.

Answer

**B.** Non-deterministic sampling.

## Prompt engineering and context crafting

**26.** You need generated rows to match an exact JSON shape. Best technique?

- A. Zero-shot prompting
- B. Few-shot prompting with example rows
- C. Lowering temperature only
- D. Opening unrelated files

Answer

**B.** Few-shot pins the format.

**27.** A first Chat answer misses null handling. Best next step?

- A. Start a new unrelated chat and retype everything.
- B. In the same thread, ask it to handle null/empty inputs and add a test.
- C. Switch languages.
- D. Repeat the prompt verbatim.

Answer

**B.** Iterate within the thread to reuse context.

**28.** Referencing `#selection` or a specific `#file` in Chat:

- A. Retrains the model.
- B. Adds that code to the prompt's context to ground the answer.
- C. Disables filtering.
- D. Increases the window size.

Answer

**B.** Explicit references inject relevant context.

**29.** "Prompt engineering" for a developer means:

- A. Fine-tuning a model with code.
- B. Crafting clear instructions and supplying context/examples to improve output.
- C. Configuring a proxy.
- D. Training a new LLM.

Answer

**B.** Instructions, context, examples — no training.

**30.** Which prompt is most likely to yield correct, well-shaped code?

- A. "Fix this."
- B. "Improve the function."
- C. "Refactor `parseConfig` to return `Result<Config, Error>`, keep the signature, add a test for a malformed file."
- D. "Make it faster somehow."

Answer

**C.** Intent + specifics + an edge case.

## Improve developer productivity

**31.** The most appropriate task for Copilot is:

- A. Guaranteeing zero vulnerabilities.
- B. Generating unit tests and edge cases you then review.
- C. Setting your product roadmap.
- D. Certifying code production-ready with no review.

Answer

**B.** Test generation is a core, well-suited use case.

**32.** A generated test passes on first run. Before relying on it you should:

- A. Nothing — passing means correct.
- B. Verify it asserts the intended behavior and covers meaningful cases.
- C. Delete other tests.
- D. Raise the temperature.

Answer

**B.** Passing ≠ correct; it can lock in a bug.

**33.** Copilot most directly reduces context switching by:

- A. Training on browser history.
- B. Explaining unfamiliar code and generating examples in the editor.
- C. Blocking the internet.
- D. Disabling extensions.

Answer

**B.** Answers and examples in place keep you in flow.

**34.** Best first step to modernize a poorly understood legacy module?

- A. Rewrite it all at once with no tests.
- B. Have Copilot explain it, then modernize incrementally with tests.
- C. Delete it.
- D. Ask for a bug-free guarantee.

Answer

**B.** Understand first, change in verified steps.

**35.** Copilot suggests replacing a concatenated SQL query with a parameterized one. Treat it as:

- A. A guaranteed fix needing no checks.
- B. A helpful security suggestion to review, test, and scan.
- C. Irrelevant to security.
- D. A reason to skip review.

Answer

**B.** Good suggestion — still validated.

## Privacy, content exclusions, and safeguards

**36.** To ensure Copilot never uses files under `secrets/` as context, use:

- A. The duplication filter
- B. Content exclusions
- C. An audit log event
- D. A prompt file

Answer

**B.** Exclusions keep paths out of context (input side).

**37.** To reduce suggestions that match public code, use:

- A. Content exclusions
- B. The duplication (public-code match) filter
- C. The REST API
- D. Audit logs

Answer

**B.** The duplication filter acts on output.

**38.** To automatically add/remove Copilot seats during onboarding, use:

- A. The Copilot REST API
- B. Content exclusions
- C. The duplication filter
- D. `.github/copilot-instructions.md`

Answer

**A.** The REST API manages seats/subscriptions.

**39.** Copilot stops suggesting in one repository. A plausible *governance* cause is:

- A. Monitor refresh rate
- B. A policy disabling the feature, or a content exclusion covering those files
- C. The color theme
- D. Model temperature

Answer

**B.** Policies (availability) and exclusions (visibility) are first to check.

**40.** Which statement about accepted suggestions is correct?

- A. GitHub owns all accepted suggestions.
- B. You generally own them but remain responsible for validation and licensing.
- C. Accepting transfers all legal responsibility to GitHub.
- D. The duplication filter guarantees license compliance.

Answer

**B.** You own and stay responsible; the filter mitigates but doesn’t guarantee.

---

## Further reading

**Source**: Study guide for Exam GH-300: GitHub Copilot

# Chapter 16 — GH-600 Exam Readiness

*Part IV — Exam readiness*

---

## In 30 seconds

- **The core idea**: a final, objective-by-objective sweep of GH-600 with a full-length practice exam.
- **Why it matters**: agentic topics are new and nuanced; a mock exam calibrates your judgment.
- **The exam angle**: a passing score is **700**; the role centers on operating and governing agents in the SDLC.
- **Remember**: agents are a fast-moving area — verify Preview vs GA before booking.

---

## 1. Objective checklist

- ☐ **Agent architecture & SDLC processes (15–20%)** — steps; anti-patterns; inputs/outputs/success criteria; planning vs execution; structured plans and validation; observability, autonomy, artifacts, human intervention.
- ☐ **Tool use & environment interaction (20–25%)** — tools and permissions; MCP servers, registries, allow lists, remote GitHub MCP; execution context; repo/branch scoping; CI invocation; autonomous PRs; error handling, retries, rollbacks, escalation, traceability.
- ☐ **Memory, state, and execution (10–15%)** — memory types and scoping; expiry/pruning/reset; durable state; resume without divergence; drift; continuity across tools.
- ☐ **Evaluation, error analysis, and tuning (15–20%)** — success criteria and signals; automated scanning; failure analysis; root-cause classification; tuning instructions/memory/tools.
- ☐ **Multi-agent orchestration (15–20%)** — orchestration patterns; isolation; conflict resolution; observability; post-hoc analysis; failure recovery; agent lifecycle.
- ☐ **Guardrails and accountability (10–15%)** — risk classification; autonomy levels; HITL; blocking violations; least privilege; authorization for irreversible changes; velocity.

---

## 2. High-yield facts

Memorize these before test day. Each is a compact, frequently tested fact from Chapters 9–14.

- **Passing score is 700.** The role centers on operating and governing agents in the SDLC, with GitHub as the system of record. Agents are fast-moving — verify Preview vs GA before booking.
- **Core safe pattern: plan → validate → act.** Configure planning distinct from execution; output a **structured plan**; prevent action until approved.
- **Every agent needs** defined **inputs, outputs, and success criteria**. No success criteria = an anti-pattern (unevaluable, untrustworthy).
- **Anti-patterns:** acting without a plan; unbounded scope; no success criteria; no artifacts; all-or- nothing autonomy.
- **Inspectable artifacts** (plan, logs, diff, draft PR) provide observability and human intervention **without slowing delivery**.
- **Tools carry permissions;** apply **least privilege** and scope to a **repo/branch**. Largest skill area is tool use & environment interaction (20–25%).
- **MCP allow lists:** enterprise **`managed-settings.json`** = **strong**, GA, non-overridable (matches by name/URL/stdio). **Custom registry** = weaker, public preview, bypassable. The **"MCP servers in Copilot"** policy governs whether MCP runs at all.
- **Cloud-agent guardrails:** only **write-access** users can trigger it; pushes to a **single branch** (`copilot/` or the PR branch) under branch protections; **limited credentials** (simple push only); opens **draft PRs** it **can't** mark ready/approve/merge; **workflows require approval** to run; the requester **can't self-approve**; **extra approval** for **unattributed** Copilot PRs; **restricted internet**; **hidden characters filtered** (prompt-injection mitigation).
- **Built-in validation: CodeQL**, dependency review (GitHub Advisory Database), **secret scanning** — **no GHAS license required**.
- **Safe execution** = error handling, **retries** (bounded), **rollbacks**, **escalation**, **traceability**.
- **Memory types: short-term** (current context), **long-term** (persists across sessions), **external** (retrieved on demand). **Scope** memory; apply **expiration/pruning/reset**.
- **State:** persist progress/decisions as **durable artifacts** to **resume** without repeating or diverging; watch for **context drift**; across tools prevent **conflicting** and **stale** context.
- **Evaluation loop:** define criteria → collect **signals** (quantitative + qualitative; scans) → analyze artifacts → **classify root cause** → tune. **Diagnose before tuning.**
- **Root-cause classes: reasoning errors**, **tool misuse**, **context/environment issues** — each maps to a fix (instructions, tools, memory/context).
- **Multi-agent conflicts: overlapping changes**, **duplicated effort**, **contradictory outputs**. **Isolation** prevents most; detect + resolve the rest. Recovery = **rollback** / **human-in-the-loop**.
- **Lifecycle:** add/update/replace/**retire** agents **without disrupting active workflows** and **preserving auditability**.
- **Guardrails: match control to risk** — automate low-risk/reversible; **HITL** for high-risk/ irreversible/compliance-sensitive; **block** policy violations. Uniform approvals are

an anti-pattern.
- **Accountability:** agent commits are **signed** and **co-authored** (Copilot + requester) with **session logs** and **audit events**; the **human** remains responsible.

---

## 3. Mock exam (GH-600)

40 questions, weighted roughly by skill area. Answer before expanding each explanation. Target 70%+ before sitting the real exam.

### Prepare agent architecture and SDLC processes

**1.** The safest architectural pattern for an agent that changes code is:

- A. Act first, explain later.
- B. Produce a structured plan, validate/approve it, then execute.
- C. Full autonomy, no plan, for speed.
- D. A human types every command.

Answer

**B.** Plan → validate → act gives a cheap human checkpoint.

**2.** Which trio must be defined before building an agent?

- A. Colors, fonts, layout
- B. Inputs, outputs, success criteria
- C. Temperature, model, seed
- D. Branch, commit, tag

Answer

**B.** The agent’s job must be specified and measurable.

**3.** You supervise an autonomous agent without watching every action by:

- A. Disabling logging.
- B. Relying on inspectable artifacts — plans, logs, diffs, draft PRs.
- C. Trusting it completely.
- D. Running it only on Fridays.

Answer

**B.** Artifacts make work reviewable and auditable.

**4.** Which is an agent anti-pattern?

- A. Scoping to a repo and branch
- B. Emitting a plan before acting
- C. Acting immediately with unbounded scope and no success criteria
- D. Producing a draft PR for review

Answer

**C.** Immediate, unbounded, unmeasurable action.

**5.** Separating planning from execution primarily lets you:

- A. Slow the agent for no reason.
- B. Validate the plan and stop the agent before any change.
- C. Disable tools permanently.
- D. Skip success criteria.

Answer

**B.** It creates a validation checkpoint.

**6.** Human intervention without slowing delivery is achieved by:

- A. Approving every step.
- B. Intervening at high-risk/irreversible points and automating the rest, using artifacts to review.
- C. Removing all oversight.
- D. Running everything manually.

Answer

**B.** Right-size intervention; automate the safe majority.

## Implement tool use and environment interaction

**7.** For strict, non-overridable MCP enforcement, use:

- A. A custom MCP registry
- B. An allow list in enterprise `managed-settings.json`
- C. A `.gitignore` entry
- D. Content exclusions

Answer

**B.** Managed settings are GA and non-overridable; a registry is weaker/bypassable.

**8.** The principle that should guide which tools an agent gets is:

- A. Grant every tool for flexibility.
- B. Least privilege — only what the task requires.
- C. Write access to all repos.
- D. Disable all tools.

Answer

**B.** Least privilege caps blast radius.

**9.** By default, when the cloud agent opens a PR, GitHub Actions workflows:

- A. Run automatically and immediately.
- B. Do not run until a user with write access approves them.
- C. Are permanently disabled.
- D. Run only if the agent approves them.

Answer

**B.** Workflows need human approval.

**10.** A robust safe-execution design for a flaky tool call is:

- A. Ignore the error and continue.
- B. Bounded retries, rollback of partial changes, escalation to a human — all logged.
- C. Retry forever.
- D. Immediately merge whatever exists.

Answer

**B.** Retries (limited), rollback, escalation, traceability.

**11.** The cloud agent is constrained in where it can write by:

- A. Pushing to any branch anywhere.
- B. Pushing only to a single branch (`copilot/` or the PR branch), under branch protections.
- C. Force-pushing to `main`.
- D. Unrestricted Git access.

Answer

**B.** Single-branch scope + protections; simple push only.

**12.** The “MCP servers in Copilot” policy controls:

- A. The color of the UI.
- B. Whether MCP servers can run at all across Copilot clients.
- C. The model temperature.
- D. The number of seats.

Answer

**B.** It gates MCP usage entirely.

**13.** Which reduces the impact of prompt injection on an agent?

- A. Broad permissions.
- B. Least privilege and scoped execution.
- C. Disabling logs.
- D. Auto-merging PRs.

Answer

**B.** Minimal permissions limit what any injection can do.

**14.** How can an agent be invoked as part of CI safely?

- A. It runs workflows automatically with no approval.
- B. It can be invoked in a workflow, but its PR’s workflows still require write-access approval.
- C. It bypasses branch protections.
- D. It merges its own PRs.

Answer

**B.** Approval gating remains in force.

## Manage memory, state, and execution

**15.** To remember a coding convention across many future sessions, use:

- A. Short-term memory
- B. Long-term memory
- C. No memory
- D. The context window alone

Answer

**B.** Long-term persists across sessions.

**16.** To resume a long task after interruption without repeating work:

- A. Start over each time.
- B. Read durable state artifacts (plan, checklist, logs).
- C. Increase temperature.
- D. Delete previous work.

Answer

**B.** Persisted state enables deterministic resume.

**17.** Context drift is:

- A. A latency metric.
- B. Gradual divergence of an agent’s behavior from the original intent over a long run.
- C. A type of MCP server.
- D. A billing model.

Answer

**B.** Divergence from intent over time.

**18.** Why apply expiration/pruning to memory?

- A. To keep every fact forever.
- B. To prevent stale context and stay within the finite window.
- C. To disable long-term memory.
- D. To increase hallucinations.

Answer

**B.** Keep memory relevant and bounded.

**19.** Sharing state across two tools, you must prevent:

- A. Conflicting and stale context
- B. Fast and slow context
- C. Public and private context
- D. Signed and unsigned context

Answer

**A.** Disagreeing sources and outdated values; use a single source of truth.

**20.** “External” memory means:

- A. Memory held only in the context window.
- B. Information retrieved on demand from a store, repo, or tool.
- C. Memory that never changes.
- D. The model’s training data.

Answer

**B.** Fetched on demand rather than held in context.

### Perform evaluation, error analysis, and tuning

**21.** An agent's PRs pass all tests but the bug persists. First step?

- A. Rewrite instructions at random.
- B. Analyze artifacts (logs, plan, traces) to find the root cause.
- C. Delete production code.
- D. Increase retries.

Answer

**B.** Diagnose before tuning.

**22.** An agent chose the wrong tool for a task. Root-cause class?

- A. Reasoning error
- B. Tool misuse
- C. Context/environment issue
- D. Network outage

Answer

**B.** Tool misuse → refine tool access/guidance.

**23.** Which is a *qualitative* evaluation signal?

- A. Number of tests passed
- B. Build duration
- C. Whether the approach is correct and readable
- D. Count of CodeQL findings

Answer

**C.** Judged, not measured.

**24.** The root cause is missing project context. Best tuning action?

- A. Add broad-permission tools.
- B. Fix memory/context — supply or scope the needed info.
- C. Lower the success criteria.
- D. Remove all instructions.

Answer

**B.** Context issue → fix context.

**25.** Why must evaluation align with development intent?

- A. Surface metrics can be satisfied while the real goal fails (e.g., weakening tests).
- B. Intent is irrelevant to agents.
- C. Qualitative signals are never useful.
- D. Tests should be deleted.

Answer

**A.** Prevent gaming a metric.

**26.** Which are legitimate automated sources of evaluation signals?

- A. CodeQL, dependency review, and secret scanning
- B. The color theme
- C. The commit message font
- D. The number of open tabs

Answer

**A.** Scans produce objective signals.

## Orchestrate multi-agent coordination

**27.** The primary way to run multiple agents in parallel safely is:

- A. Give them all the same branch.
- B. Isolate each with its own scope/context (e.g., separate branches).
- C. Disable logging.
- D. Run them one at a time.

Answer

**B.** Isolation prevents collisions.

**28.** Which is a multi-agent conflict type?

- A. Overlapping code changes
- B. A slow network
- C. A large context window
- D. A signed commit

Answer

**A.** Plus duplicated effort and contradictory outputs.

**29.** A stalled agent in a multi-agent workflow should be handled by:

- A. Ignoring it and merging the rest.
- B. Detecting the stall and applying rollback or human-in-the-loop.
- C. Deleting the repository.
- D. Granting more permissions.

Answer

**B.** Detect degraded state; recover.

**30.** Post-hoc analysis of a multi-agent run requires:

- A. Turning off logging.
- B. Documented decisions, handoffs, and outcomes as auditable artifacts.
- C. Relying on agent memory only.
- D. Merging without review.

Answer

**B.** Artifacts enable reconstruction.

**31.** Retiring an agent from a running workflow requires:

- A. Doing it instantly regardless of active work.

- B. Retiring without disrupting active workflows and preserving auditability.
- C. Deleting its past logs first.
- D. Distributing its permissions to all agents.

Answer

**B.** Preserve continuity and history.

**32.** An orchestrator/worker pattern is:

- A. A billing tier.
- B. A lead agent delegating sub-tasks to specialized agents.
- C. A type of branch protection.
- D. A memory store.

Answer

**B.** Coordination via delegation.

### Implement guardrails and accountability

**33.** How much autonomy to grant an action is governed by:

- A. Full autonomy for everything.
- B. Matching the control to the action’s risk.
- C. Human approval for every action.
- D. The time of day.

Answer

**B.** Automate low-risk; HITL for high-risk/irreversible.

**34.** Can the cloud agent merge its own PR by default?

- A. Yes, automatically.
- B. No — it opens a draft PR a human must review and merge.
- C. Yes, if tests pass.
- D. Only on weekends.

Answer

**B.** It can’t mark ready, approve, or merge.

**35.** Why does GitHub prevent the requester from approving the agent’s PR (when approval is required)?

- A. To slow that developer down.
- B. To preserve separation-of-duties / required-approval controls.
- C. Approvals are disabled for agents.
- D. The agent approves instead.

Answer

**B.** Preserves required-approval controls.

**36.** Uniform human approval for every agent action is:

- A. Best practice.
- B. An anti-pattern — it kills velocity without targeting real risk.

- C. Required by GitHub.
- D. Impossible.

Answer

**B.** Approve the irreversible; automate the safe.

**37.** Accountability for an autonomous agent's work on GitHub is provided by:

- A. The agent being legally responsible.
- B. Signed, co-authored commits + session logs + audit events; humans remain accountable.
- C. No traceability at all.
- D. Only the model provider being accountable.

Answer

**B.** Traceable work; human accountability.

**38.** An extra approval is required when a Copilot PR:

- A. Passes tests.
- B. Isn't attributed to a person (unattributed).
- C. Is small.
- D. Targets a feature branch.

Answer

**B.** Unattributed Copilot PRs need an additional approval (where approvals are required).

**39.** Which action most clearly warrants human-in-the-loop?

- A. Reading an issue.
- B. Deploying to production (irreversible/high-risk).
- C. Listing open PRs.
- D. Formatting a file.

Answer

**B.** Irreversible, high-risk actions need HITL.

**40.** GitHub's coding agent validates generated code using:

- A. CodeQL, dependency review, and secret scanning — no GHAS license required.
- B. Only manual review.
- C. Nothing by default.
- D. A separate paid product only.

Answer

**A.** Built-in security validation without a GHAS license.

---

## Further reading

**Source**: Study guide for Exam GH-600: Developing in Agentic AI Systems

# Annex A — Glossary

*Annexes*

Consolidated definitions from the chapters, one wording per term, alphabetized. Chapter references point to where the term is introduced.

---

- **Agent** — an AI system that pursues a goal by planning steps, taking actions through tools, observing results, and iterating with some autonomy, rather than producing a single response. (Ch. 9)
- **Agent isolation** — giving each agent its own context, scope, and workspace (e.g., separate branches) so parallel agents don't interfere; the precondition for safe parallelism. (Ch. 13)
- **Agent mode** — a Copilot capability where it works autonomously toward a goal: deciding which files to change, running tools (build, tests), and iterating until done or blocked. (Ch. 6)
- **Allow list (MCP)** — an administrator-defined list of approved MCP servers; prefer enterprise `managed-settings.json` for strong, non-overridable enforcement over a custom registry. (Ch. 10)
- **Autonomy level** — the degree of independent action granted to an agent for a class of actions, assigned by classifying actions by operational, security, and compliance risk. (Ch. 14)
- **Content drift** — see **Context drift**.
- **Content exclusion** — an admin setting that prevents specified files or repositories from being used as context by Copilot; they are never sent to the model (input side). (Ch. 8)
- **Context** — everything the model "sees" for a request: your words plus the code Copilot gathers (current file, selection, neighboring tabs) and anything you deliberately reference. (Ch. 3)
- **Context drift** — the gradual divergence of an agent's behavior from the original intent during a long run. (Ch. 11)
- **Context switching** — the productivity cost of leaving a task to look something up elsewhere; Copilot reduces it by answering and generating examples in place. (Ch. 7)
- **Context window** — the maximum number of tokens (prompt + response) a model can process in one request. (Ch. 1)
- **Copilot Edits** — an IDE capability where you describe a change and Copilot proposes edits across multiple files you scope, which you accept or discard as a set. (Ch. 6)
- **Duplication detection filter (public-code match)** — an optional setting that suppresses suggestions matching public code on GitHub (output side); an IP/licensing safeguard. (Ch. 8)

- **Evaluation signal** — an observable indicator of agent performance; quantitative (measurable) or qualitative (judged). (Ch. 12)
- **Few-shot prompting** — including one or more examples of the desired input/output or style so the model imitates the pattern. (Ch. 3)
- **GitHub Copilot CLI** — a command-line tool that brings a Copilot agent into your terminal to answer questions, write/debug code, run tasks, and interact with GitHub.com. (Ch. 5)
- **Guardrail** — a constraint that limits what an agent may do: a permission boundary, a required approval, a blocked action, an execution scope. (Ch. 14)
- **Hallucination (fabrication)** — a confident, plausible-sounding output that is factually wrong; inherent to probabilistic generation. (Ch. 1)
- **Human-in-the-loop (HITL)** — a checkpoint requiring explicit human approval before an agent proceeds; reserved for high-judgment or irreversible actions. (Ch. 14)
- **Inline suggestion** — grey "ghost text" Copilot proposes at your cursor as you type. (Ch. 5)
- **Inspectable artifact** — a durable, reviewable output an agent produces (plan, session log, diff, draft PR) that enables supervision. (Ch. 9)
- **Instructions file** — a Markdown file of standing guidance Copilot applies automatically (`.github/copilot-instructions.md`; path-specific `*.instructions.md`; `AGENTS.md`). (Ch. 6)
- **Large language model (LLM)** — a neural network trained to predict the next token given prior tokens; the engine behind Copilot. (Ch. 1)
- **Least privilege** — granting the minimum permissions and narrowest execution scope an action requires, to cap blast radius. (Ch. 10, 14)
- **Memory (agent)** — information an agent retains: short-term (current context), long-term (persists across sessions), or external (retrieved on demand). (Ch. 11)
- **Model Context Protocol (MCP)** — an open standard for connecting AI agents to external tools and data sources through "MCP servers." (Ch. 6, 10)
- **MCP server** — a server implementing MCP that exposes tools and data to an agent in a standard, discoverable way; the GitHub MCP server is built in. (Ch. 10)
- **Orchestration pattern** — the structure that coordinates multiple agents (orchestrator/worker, sequential, parallel) toward a shared goal. (Ch. 13)
- **Prompt (Copilot)** — the bundle of context Copilot assembles and sends to the model, more than what you typed. (Ch. 2)
- **Prompt engineering** — designing and refining instructions and context to get more accurate, relevant, useful output. (Ch. 3)
- **Prompt file** — a reusable, parameterizable prompt saved in the repository so a team runs the same request consistently. (Ch. 6)
- **Proxy** — a GitHub-operated service between your editor and the model where filtering and processing are applied. (Ch. 2)
- **Responsible AI** — building and using AI in ways that are fair, reliable & safe, private & secure, inclusive, transparent, and accountable. (Ch. 4)
- **Root-cause classification** — sorting a failure into reasoning error, tool misuse, or context/ environment issue so the fix targets the real problem. (Ch. 12)
- **State (agent)** — the record of where the agent is in its task (progress, decisions, remaining work), persisted as durable artifacts to enable resume. (Ch. 11)
- **Structured plan** — an explicit, inspectable list of intended steps produced before acting. (Ch. 9)
- **Sub-agent** — a specialized agent a primary agent delegates a sub-task to, running in its own context to optimize token usage. (Ch. 6)

- **Success criteria** — the explicit, testable conditions that define a successful agent outcome. (Ch. 9, 12)
- **Token** — the unit an LLM reads and writes, roughly a word fragment; models have a finite context measured in tokens. (Ch. 1)
- **Tool** — a capability an agent can invoke to affect the world (read a file, run a command, open a PR), carrying permissions. (Ch. 10)
- **Zero-shot prompting** — asking for a result with no example, relying on the model's general training. (Ch. 3)

# Annex B — Product & feature reference

*Annexes*

Quick reference "cards" for the products and features tested by GH-300 and GH-600. Each notes what it is, when to use it, and the chapter it maps to.

---

## GitHub Copilot surfaces

- **Inline suggestions** — grey ghost-text completions as you type; accept with Tab. Best for in-flow coding. (Ch. 5)
- **Copilot Chat** — conversational help with slash commands (`/explain`, `/fix`, `/tests`) and context references (`#file`, `#selection`). Best for explaining/refactoring specific code. (Ch. 3, 5)
- **GitHub Copilot CLI** — standalone `copilot` terminal agent; interactive or programmatic (`-p`); plan mode; can edit files, run commands, open PRs; asks approval; run from trusted directories. (Ch. 5)
- **Agent mode** — autonomous, goal-driven multi-file changes that run tools (tests/build) and iterate. (Ch. 6)
- **Copilot Edits** — described multi-file edits you scope and accept/discard as a set. (Ch. 6)

## Customization & extensibility

- **Instructions files** — standing standards: `.github/copilot-instructions.md` (repo-wide); path-specific `.github/instructions/*.instructions.md` with `applyTo`; `AGENTS.md` (agent). Precedence: personal > repo > org. (Ch. 6)
- **Prompt files** — reusable, repeatable requests saved in the repo. (Ch. 6)
- **Model Context Protocol (MCP)** — open standard to connect agents to external tools/data; GitHub MCP server built in. (Ch. 6, 10)
- **Custom agents / sub-agents** — specialized agents (e.g., Explore, Task, Code review) the model can delegate to; defined in Markdown at user/repo/org level. (Ch. 6)
- **Spaces** — curated, reusable/shareable context. (Ch. 6)
- **Spark** — build an app from a description. (Ch. 6)

## Administration & safeguards

- **Organization policies** — enable/disable features and set availability across IDEs and github.com. (Ch. 8)
- **Copilot Code Review policies** — govern availability of Copilot's PR review. (Ch. 6, 8)
- **Content exclusions** — keep specified files/paths out of Copilot's context (input side). (Ch. 8)
- **Public-code duplication filter** — suppress suggestions matching public code (output side). (Ch. 8)
- **Audit log events** — records of Copilot admin/usage events; answer "who did what." (Ch. 8)
- **Copilot REST API** — manage seats/subscriptions programmatically. (Ch. 8)

## MCP governance (GH-600)

- **"MCP servers in Copilot" policy** — whether MCP servers may run at all. (Ch. 10)
- **MCP allow list — managed settings** — enterprise `managed-settings.json`; GA, strong, non-overridable (matches by name/URL/stdio). Preferred. (Ch. 10)
- **MCP allow list — custom registry** — self-hosted; public preview; weaker/bypassable. (Ch. 10)

## Agentic building blocks (GH-600)

- **Structured plan** — explicit steps produced before acting; enables validation. (Ch. 9)
- **Agent memory** — short-term / long-term / external; scope and apply expiry/pruning/reset. (Ch. 11)
- **Evaluation signals & scanning** — quantitative/qualitative signals; CodeQL, dependency review, secret scanning as sources. (Ch. 12)
- **Multi-agent orchestration** — patterns, isolation, conflict resolution, recovery, lifecycle. (Ch. 13)
- **Guardrails / autonomy levels / HITL** — match control to risk; least privilege; block violations. (Ch. 14)

## Cloud-agent guardrails at a glance (GH-600)

- Only **write-access** users trigger it; pushes to a **single branch**; **limited credentials** (simple push). Opens **draft PRs** it can't approve/merge; **workflows need approval**; requester can't self-approve; **extra approval** for unattributed PRs; **restricted internet**; **CodeQL + secret scanning** (no GHAS license); **signed, co-authored commits** + session logs + audit events. (Ch. 10, 14)

# Annex C — Objective-to-chapter map

*Annexes*

Full traceability: every skill area of GH-300 and GH-600 maps to a chapter.

---

## GH-300 — GitHub Copilot (skills measured as of August 7, 2026)

| Skill area | Weight | Chapter(s) |
|---|---|---|
| Use GitHub Copilot responsibly | 15–20% | 4 |
| Use GitHub Copilot features | 25–30% | 5, 6 (org policies: 8) |
| Understand GitHub Copilot data and architecture | 10–15% | 1, 2 |
| Apply prompt engineering and context crafting | 10–15% | 3 |
| Improve developer productivity with GitHub Copilot | 10–15% | 7 |
| Configure privacy, content exclusions, and safeguards | 10–15% | 8 |

## GH-600 — Developing in Agentic AI Systems

| Skill area | Weight | Chapter(s) |
|---|---|---|
| Prepare agent architecture and SDLC processes | 15–20% | 9 |
| Implement tool use and environment interaction | 20–25% | 10 |
| Manage memory, state, and execution | 10–15% | 11 |
| Perform evaluation, error analysis, and tuning | 15–20% | 12 |
| Orchestrate multi-agent coordination | 15–20% | 13 |
| Implement guardrails and accountability | 10–15% | 14 |

---

**Source**: GH-300 study guide · GH-600 study guide

# Annex D — Further resources

*Annexes*

Consolidated, cited sources. Every **Source** used in the chapters is collected here. Populate and verify as chapters are written.

---

## Official study guides

- Study guide for Exam GH-300: GitHub Copilot
- Study guide for Exam GH-600: Developing in Agentic AI Systems

## Microsoft Learn training

- GitHub Copilot Fundamentals — Part 1
- GitHub Copilot Fundamentals — Part 2
- Responsible AI with GitHub Copilot
- Develop unit tests using GitHub Copilot tools
- Developer use cases for AI with GitHub Copilot
- Foundations of Agentic AI in GitHub

## GitHub documentation

- GitHub Copilot documentation
- Copilot plans and features
- GitHub Copilot in the CLI
- Copilot agent mode
- Extending Copilot with MCP
- Configuring and auditing content exclusion
- GitHub Trust Center (data handling)

## Exam logistics

- Exam sandbox
- Exam scoring and score reports