GitHub / Microsoft Certification

GH-600

Developing in Agentic AI Systems

A detailed study guide for operating, supervising, evaluating, and governing AI agents inside the software development lifecycle, using GitHub as the system of record and control plane. Built from the official skills outline, GitHub Docs, VS Code docs, and Microsoft Learn.

15–20% Agent architecture and SDLC	**20–25%** Tool use and environment	**10–15%** Memory, state, execution	**15–20%** Evaluation and tuning	**15–20%** Multi-agent coordination	**10–15%** Guardrails and accountability

Edition: October 2026. Independent study material. Practice items are original and are not real exam questions. GitHub Copilot features, names, and preview status change frequently: confirm details against GitHub Docs before exam day.

Contents

START HERE — **The GH-600 exam and how to use this guide**

0.1 Who the exam is for
0.2 Skills measured and weighting
0.3 Exam mechanics
0.4 How this guide is organised
0.5 Four-week study plan

FOUNDATION — **The GitHub agent landscape**

F.1 Where agents run
F.2 Customisation files
F.3 Built-in tool aliases
F.4 GitHub controls agents inherit

DOMAIN 1 — **Prepare agent architecture and SDLC processes**

1.1 Integrate agents into the software development lifecycle
1.2 Define boundaries between planning, reasoning, and action
1.3 Configure observability and control for autonomous agents
1.4 Additional practice questions

DOMAIN 2 — **Implement tool use and environment interaction**

2.1 Select and configure agent tools
2.2 Configure MCP servers
2.3 Integrate agents within development environments
2.4 Operate agents with safe execution paths and robust error handling
2.5 Key terms
2.6 Additional practice questions

DOMAIN 3 — **Manage memory, state, and execution**

3.1 Implement agent memory strategies
3.2 Persist agent state and manage context drift
3.3 Ensure continuity of agent memory and state across tools and environments
3.4 Additional practice questions

DOMAIN 4 — **Perform evaluation, error analysis, and tuning**

4.1 Define success criteria and evaluation signals for agent tasks
4.2 Analyze agent failures and identify root causes
4.3 Tune agent behavior based on evaluation results
4.4 Key terms
4.5 Additional practice questions

DOMAIN 5 — **Orchestrate multi-agent coordination**

5.1 Operate and manage multi-agent workflows
5.2 Configure observability for multi-agent behavior
5.3 Detect and respond to multi-agent failures and degraded behavior
5.4 Manage the lifecycle of agents within multi-agent workflows
5.5 Key terms
5.6 Additional practice questions

DOMAIN 6 — **Implement guardrails and accountability**

6.1 Define autonomy levels
6.2 Implement guardrails and human-in-the-loop workflows

6.3 Accountability: attribution, audit and retention
6.4 Key terms
6.5 Additional practice questions

CROSS-DOMAIN PRACTICE — Case study practice

C.1 A method for case studies
C.2 Case study: Fabrikam Payments

EXAM DAY — Exam technique and signal words

E.1 Signal words that map to answers
E.2 Distractor patterns
E.3 Time management

APPENDIX A — One-page cheat sheet

APPENDIX B — Glossary

APPENDIX C — Hands-on lab plan

APPENDIX D — Final readiness checklist

APPENDIX E — Official resources

START HERE

The GH-600 exam and how to use this guide

Orientation, exam mechanics, and a study plan

GH-600 measures whether you can run AI agents as reliable participants in a production software process. It is not an exam about prompting a chat window. It asks how you scope an agent's work, give it exactly the tools it needs, keep its state durable, judge its output with real signals, coordinate several agents without collisions, and decide where humans must stay in control. GitHub is the backbone throughout: issues define work, branches and pull requests carry it, Actions runs it, rulesets and environments gate it, and audit logs record it.

700	**6**	**GitHub**	**Annual**
Passing score on a 1–1000 scaled score	Skill domains in the outline	System of record and control plane	Free online renewal assessment

0.1 Who the exam is for

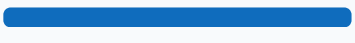
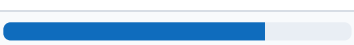
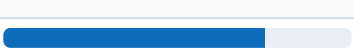
The official audience profile describes someone with subject matter expertise in **operating, integrating, supervising, and governing AI agents inside production-grade SDLC workflows and development environments**, ensuring reliability, safety, and velocity with GitHub as the system of record and control plane. Responsibilities include operating agent workflows inside the SDLC, supervising autonomous behaviour with GitHub controls, evaluating and tuning agent outputs using scans and artifacts, configuring custom agents, and coordinating multi-agent execution safely.

You are expected to know the SDLC, GitHub workflows and controls, code quality, security, and review practices, and to have hands-on experience with coding agents including GitHub Copilot, MCP servers, and agent customisation such as custom instructions, custom agents, tools, and Copilot setup steps. You work with architects, platform engineers, DevOps engineers, developers, product managers, and security engineers.

The exam rewards three kinds of judgement:

- **Configuration literacy.** Read an agent profile, an MCP configuration, a workflow file, or a hooks file and say what it allows, what it blocks, and what is missing.
- **Least privilege by default.** Choose the narrowest tool list, token permission, repository scope, or MCP allowlist that still lets the agent finish the job.
- **Right-sized human control.** Put approvals where they materially reduce risk (irreversible, security-sensitive, compliance-sensitive changes) and remove them where they only slow delivery.

0.2 Skills measured and weighting

Domain	Weight	Relative emphasis	Chapter
Prepare agent architecture and SDLC processes	15–20%		Domain 1
Implement tool use and environment interaction	20–25%		Domain 2
Manage memory, state, and execution	10–15%		Domain 3
Perform evaluation, error analysis, and tuning	15–20%		Domain 4
Orchestrate multi-agent coordination	15–20%		Domain 5

Domain	Weight	Relative emphasis	Chapter
Implement guardrails and accountability	10–15%		Domain 6

EXAM TIP

Domain 2 (tools, MCP, environments, error handling) is the single largest domain, and its content reappears inside Domains 5 and 6. Study it immediately after the orientation chapter.

0.3 Exam mechanics

Item	What to expect
Scoring	Scaled score from 1 to 1000; 700 or greater is required to pass. No penalty for wrong answers, so answer every item.
Preview features	Most questions cover generally available features, but commonly used preview features may appear. Several agent capabilities (for example some custom agent surfaces) are in public preview.
Localisation	English is updated first; localised versions follow about eight weeks later. If your language is unavailable you can request 30 extra minutes.
Sandbox	Rehearse the interface at aka.ms/examdemo.
Renewal	Microsoft certifications in this family expire annually and renew through a free online assessment.

Question formats

Format	What it looks like in GH-600	Technique
Multiple choice	A scenario and one best action.	Eliminate options that grant more access than required or add approvals that do not reduce risk.
Multiple response	"Which two..." for paired controls (for example a hook plus an environment reviewer).	Check whether options are parts of one solution or alternatives.
Yes/No series	Three statements about a configuration file or log excerpt.	Judge each statement against the file, not against what you expect the file to say.
Code / configuration completion	YAML front matter, JSON MCP config, or workflow YAML with blanks.	Find the requirement word ("read-only", "must not run commands", "hidden from picker") and map it to one key.
Hot area	Choose the right value for each drop-down in a sentence or file.	Each blank is usually a contrasting pair (allow vs deny, job vs workflow scope).
Case study	A company scenario with requirements across domains.	Read requirements first and number them; you cannot return after leaving.

0.4 How this guide is organised

- **Agent landscape orientation** maps every GitHub Copilot surface, customisation file, and control you will meet.
- **Domains 1 to 6** follow the official outline bullet by bullet: objective map, concepts, decision tables, verified configuration listings, callouts, original practice questions with answers, and key takeaways.

- **Case study practice** gives a method and a full multi-requirement case.
- **Exam technique** lists signal words that map directly to answers and common distractor patterns.
- **Appendices** contain a cheat sheet, glossary, hands-on lab plan, readiness checklist, and official resources.

EXAM TIP

Patterns that frequently decide an answer.

COMMON TRAP

Plausible distractors and misconceptions.

PRODUCTION NOTE

What changes when you run agents for real across a team or enterprise.

0.5 Four-week study plan

Assumes 8 to 10 hours per week, a GitHub organisation (or a personal account with a Copilot plan that includes cloud agent), VS Code with GitHub Copilot, and GitHub Copilot CLI.

Week	Read	Hands-on	Checkpoint
1	Orientation; Domains 1 and 2	Write `.github/copilot-instructions.md` and a planner custom agent with read-only tools; assign a well-scoped issue to Copilot cloud agent; add a `copilot-setup-steps.yml`; configure a repository MCP server and the GitHub remote MCP server in VS Code; run Copilot CLI with `--allow-tool` and `--deny-tool`.	Domain 1 and 2 practice questions at 80%.
2	Domains 3 and 4	Enable Copilot Memory on a test repository; use a progress file to resume work; pass a plan between Actions jobs with outputs and artifacts; make CodeQL and unit tests required checks; read a cloud agent session log end to end and classify a failure.	Domain 3 and 4 practice questions at 80%.
3	Domains 5 and 6	Build an orchestrator agent with two subagents and handoffs; run two agents in parallel on separate branches; add a hooks file that blocks destructive commands; protect a deploy job with an environment and required reviewers; query the audit log for agent activity.	Domain 5 and 6 practice questions at 80%.
4	Case study; Exam technique; Appendices	Repeat weak labs. Sit any official practice assessment available for GH-600 under timed conditions.	85% or better on all practice sets.

FOUNDATION

The GitHub agent landscape

Surfaces, customisation files, and controls used in every domain

Almost every GH-600 item assumes you can place an agent in one of GitHub Copilot's surfaces and know which file configures it and which control constrains it. This chapter is the map.

F.1 Where agents run

Surface	Execution context	Typical use and key facts
Copilot cloud agent (formerly Copilot coding agent)	Ephemeral cloud development environment powered by GitHub Actions	Delegated, asynchronous work from an issue, the agents panel, chat, or a PR. Handles branch creation, commits, and pushes; opens one pull request per task. Changes only the repository where the task started; one branch at a time; each session has a maximum execution time of 59 minutes (shorter via `timeout-minutes` in `copilot-setup-steps.yml`).
VS Code agent mode and custom agents	Your local workspace	Interactive, developer-directed work with tool approvals, subagents, and handoffs between custom agents.
GitHub Copilot CLI	Your terminal, or a CI runner	Interactive or programmatic agent runs; tool permissions via allow and deny rules; autopilot for unattended runs; can delegate to cloud agent.
Copilot code review	GitHub.com pull requests	Automated review feedback and suggested fixes, guided by repository instructions; can be requested or configured to run automatically.
Copilot automations and GitHub Agentic Workflows	GitHub.com, GitHub Actions	Scheduled or event-driven agent tasks (for example triage or weekly release notes). Agentic Workflows are written in Markdown and run as Actions workflows.

COMMON TRAP

Cloud agent cannot make changes across multiple repositories in one run, and by default its GitHub MCP server only sees context in the repository it is working in. A requirement that spans repositories needs several tasks or broader MCP configuration, not a bigger prompt.

F.2 Customisation files

File or location	Purpose
`.github/copilot-instructions.md`	Repository-wide custom instructions: build commands, conventions, test expectations. Long-lived "memory" you control.
`.github/instructions/*.instructions.md`	Path-specific instructions with an `applyTo` glob.
`AGENTS.md`	Agent instructions file recognised by Copilot agents.
`.github/agents/*.agent.md`	Custom agent profiles: YAML front matter (`description` required; `name` , `tools` , `model` , `target` , `mcp-servers` , `disable-model-invocation` , `user-invocable`) plus a Markdown prompt of up to 30,000 characters. VS Code adds `handoffs` and subagent settings.

File or location	Purpose
`.github/workflows/copilot-setup-steps.yml`	Prepares cloud agent's environment (dependencies, tools, runner, timeout) before work starts.
Repository MCP configuration (Settings)	MCP servers available to cloud agent; secrets supplied through the repository's Copilot environment.
`.vscode/mcp.json`	MCP servers for VS Code in this workspace.
Hooks configuration	Commands that run at agent lifecycle events (for example before a tool runs) for validation, blocking, or logging.
Agent skills and prompt files	Reusable task-specific instructions and resources.

F.3 Built-in tool aliases

Custom agent `tools` lists use case-insensitive aliases. Omitting `tools` or using `["*"]` enables everything; `[]` disables all tools; unrecognised names are ignored. MCP tools are referenced as `server/tool` or `server/*`.

Alias	Compatible aliases	Grants
`read`	Read, NotebookRead	Read file contents.
`search`	Grep, Glob	Search for files or text.
`edit`	Edit, MultiEdit, Write, NotebookEdit	Change files.
`execute`	shell, Bash, powershell	Run shell commands.
`agent`	custom-agent, Task	Invoke another custom agent.
`web`	WebSearch, WebFetch	Fetch URLs and search the web (not applicable to cloud agent).
`todo`	TodoWrite	Structured task lists (VS Code; not cloud agent).

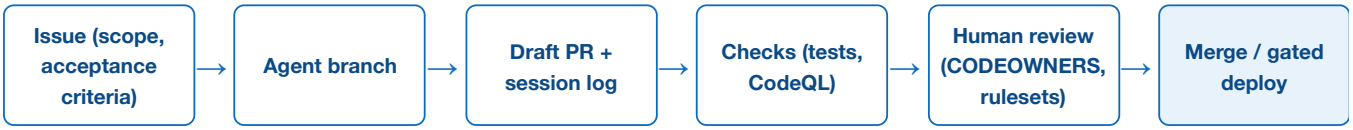
Listing F.1: A read-only planning agent (least privilege)

```
---
name: implementation-planner
description: Analyses the code base and writes a step-by-step implementation plan. Never edits files.
tools: ["read", "search", "github/*"]
disable-model-invocation: true
---
You are a planning agent. Produce a Markdown plan with: goal, affected files,
ordered steps, tests to add, risks, and open questions. Do not modify files.
```

EXAM TIP

"Must not edit files or run commands" removes `edit` and `execute`. "Must not delegate" also removes `agent`, because a subagent could hold the forbidden tools. Listing an MCP tool in `tools` does not install or authenticate the server; the server must also be configured.

F.4 GitHub controls agents inherit



Control	What it constrains for agents	Domain
Tool lists, CLI allow/deny, approvals	Which actions the agent can attempt	D2, D6
MCP registry and allowlist policies	Which external servers agents may use	D2
`GITHUB_TOKEN` permissions	What a workflow (including agent-invoking ones) can write	D2, D6
Rulesets, branch protection, required checks	What can merge, and on what evidence	D4, D6
CODEOWNERS and required reviewers	Who must approve sensitive paths	D6
Environments with required reviewers	Deployments that need a human gate	D6
Hooks	Deterministic validation, blocking, or logging during agent work	D6
Content exclusion	Files Copilot must not use	D6
Session logs, Actions logs, artifacts, audit log	Traceability and post-hoc analysis	D1, D4, D5, D6
Copilot Memory, instructions files, progress files	What the agent remembers and how state survives	D3

References

- Study guide for Exam GH-600: Developing in Agentic AI Systems
- GitHub Copilot on GitHub.com
- Custom agents configuration
- Foundations of Agentic AI in GitHub (Microsoft Learn module)

DOMAIN 1

Prepare agent architecture and SDLC processes

Exam weight: 15–20%

This domain tests whether you can place AI coding agents inside a real software development lifecycle without losing control of quality, scope, or accountability. The exam expects you to know *where* each GitHub Copilot surface fits (Copilot cloud agent, Copilot code review, agent mode in VS Code, Copilot CLI), *what* an agent needs as input and must produce as output, *how* to keep planning separate from execution, and *which* GitHub-native artifacts and controls let humans supervise autonomous work without becoming a bottleneck.

Two ideas from the official GH-600 training run through every objective. First, GitHub is both the **system of record** (issues, branches, commits, pull requests, workflow runs, reviews) and the **control plane** (required reviews, required checks, CODEOWNERS, rulesets, environments). Second, **agents propose; humans and policy accept**. Study each section with those two lenses: for every configuration, ask what evidence it produces and which control enforces it. Expect scenario questions that contrast a natural-language instruction ("do not edit files") with a real enforcement mechanism (a read-only tool list, plan mode, a required check). The enforcement mechanism almost always wins.

Skill group (official outline)	Objective	Section
Integrate agents into the software development lifecycle (SDLC)	Identify steps for agents to perform	1.1.1
	Identify and mitigate common anti-patterns in agents	1.1.2
	Define inputs, outputs, and success criteria for agents	1.1.3
Define boundaries between planning, reasoning, and action	Configure agent planning to be distinct from agent execution	1.2.1
	Configure an agent to output a structured plan	1.2.2
	Validate agent plans	1.2.3
	Prevent agent action until the agent checked and approved	1.2.4
Configure observability and control for autonomous agents	Plan and implement the degree of agent autonomy, including guardrails	1.3.1
	Configure agent to produce inspectable artifacts within standard development tooling	1.3.2
	Configure human intervention for autonomous agents without slowing delivery	1.3.3

1.1 Integrate agents into the software development lifecycle

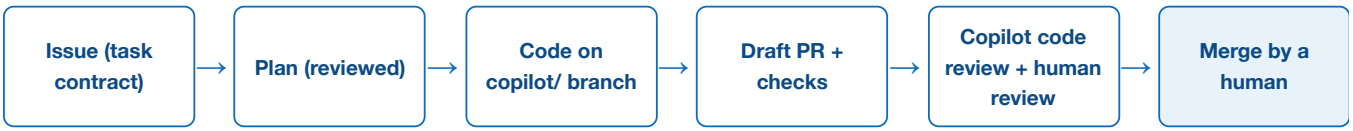
The GH-600 training module *Designing Agent Architecture and SDLC Integration* warns against treating an agent as a general-purpose developer that roams across the whole lifecycle. When an agent has no defined stage, it becomes hard to reason about its behaviour, limit its impact, or audit what it did. The reliable approach is to map each agent responsibility to a lifecycle stage where GitHub can enforce a boundary, then define the task contract (inputs, outputs, success criteria) for that stage.

1.1.1 Identify steps for agents to perform

A simplified SDLC has four stages: planning, implementation, validation and deployment. Each stage maps to a GitHub surface where work and evidence are recorded. The table below follows the mapping used in the official training and adds the Copilot surface you would normally use at each step.

SDLC stage	Typical agent responsibility	Primary GitHub artifact	Copilot surface that usually fits
Planning	Research the code, draft scope, propose steps, define success criteria	Issues, PR description or comments, Agents tab sessions, plan files	VS Code Plan mode or a planner custom agent; Copilot CLI plan mode; cloud agent research and plan session on GitHub.com
Implementation	Create a branch, edit files, commit, open or update a pull request	Branch, commits, pull request	Copilot cloud agent (background, issue or prompt driven); agent mode in VS Code; Copilot CLI (interactive or autopilot)
Validation	Run builds, tests, linters and scans; iterate on failures; review the diff	Workflow runs, checks, artifacts, review comments	Cloud agent's own ephemeral environment; Copilot code review on the PR; GitHub Actions required checks
Deployment	Usually restricted: the agent prepares, a human approves	Environments and deployment approvals	Agents prepare changes; environments with required reviewers gate execution

Translated into an end-to-end GitHub flow, a typical agentic pipeline looks like this:



Issue to plan. The issue is the agent's prompt. GitHub's best-practice guide for the cloud agent says to think of an assigned issue *as a prompt* and to check whether it would work as one. Planning can happen in VS Code (Plan mode, `/plan`), in Copilot CLI (`/plan`, `Shift+Tab`, or `--plan`), or on GitHub.com, where a cloud agent session started from a prompt can research the repository and propose a plan before any code is written.

Plan to code. Copilot cloud agent works in an ephemeral development environment powered by GitHub Actions, where it can explore code, make changes, and run tests and linters. Assigning an issue to Copilot always creates a pull request; starting a task with a prompt from the agents tab, agents panel, dashboard or Copilot Chat (`/task`) works on a branch by default, so you can review the diff, steer and iterate before you click **Create pull request**. Locally, agent mode in VS Code and Copilot CLI make the same kind of multi-step edits on your machine under your approval settings. Copilot CLI can also hand a task to the cloud agent with `/delegate` (or a prompt prefixed with `&`), which creates a branch, opens a draft pull request and works in the background.

Code to PR. The pull request is the unit of contribution. The cloud agent pushes only to its own branch (a new `copilot/` branch, or the existing PR branch when triggered by an `@copilot` mention on a PR), keeps the PR title and body up to date as it works, and requests your review when it finishes.

PR to review. By default the cloud agent already runs CodeQL, checks newly added dependencies against the GitHub Advisory Database, runs secret scanning, and gets a second opinion from Copilot code review before it completes the PR. You can additionally request Copilot code review on any PR or configure automatic reviews (user settings or rulesets), so an automated review lands before a human reviewer looks at the change.

Review to merge. Merging stays human. The cloud agent cannot mark its PR as ready for review, cannot approve it, and cannot merge it. The person who asked Copilot to create the PR cannot approve it either, so the "Required approvals" rule still requires an independent reviewer.

Need	Best fit	Why
Well-scoped backlog issue, work should continue while you do something else	Assign the issue to Copilot cloud agent	Runs in the background on GitHub Actions, opens a PR and requests review; fully traceable on GitHub
Agree an approach before any code exists, then iterate on a branch	Cloud agent session started from a prompt (research, plan, iterate)	Works on a branch by default; you create the PR only when satisfied
Interactive, multi-file change where you want to watch every tool call	Agent mode in VS Code (Agent role) with manual permissions	Local, synchronous, approvals per tool call; diff reviewed in the editor

Need	Best fit	Why
Terminal-centric workflow, scripting, or quick hand-off to the cloud	Copilot CLI (interactive, `-p`, `/delegate`)	Plan mode, autopilot, permission flags, and delegation to the cloud agent from one tool
Automated first-pass review of every PR before human review	Copilot code review, automatic via ruleset	Runs on PR creation (and optionally on new pushes and drafts); comments and suggested fixes
Recurring repository task on a schedule or event	Copilot automations (from the repository's **Agents** tab)	Defines the task once; GitHub restricts tools and ignores events from users without write access by default

Not every task should go to an agent. GitHub's guidance suggests starting with simpler tasks (bug fixes, UI tweaks, test coverage, documentation, accessibility, technical debt) and keeping the following with humans or with heavier supervision: broad cross-repository refactoring, work that needs deep domain knowledge or substantial business logic, production-critical issues, security, PII and authentication work, incident response, ambiguous or open-ended tasks, and learning tasks where the developer wants the understanding.

EXAM TIP

Match the stage to the tool. "Draft implementation from an issue and have an automated review before a human" means **assign the issue to Copilot cloud agent** and **enable Copilot code review on the PR**. Plan mode does not open pull requests, and Copilot code review runs on pull requests, not on issues.

NOTE

Copilot cloud agent was formerly called Copilot coding agent. Older material, and some exam items, may still use the former name. Likewise, VS Code custom agents were previously called custom chat modes (`.chatmode.md` files are renamed to `.agent.md`), and the old VS Code Edit mode is deprecated in favour of the Agent role or a custom agent.

1.1.2 Identify and mitigate common anti-patterns in agents

The *Foundations of Agentic AI in GitHub* module lists four early-stage failure modes: **planless execution**, **over-permissioned agents**, **hidden reasoning** and **blind trust in automation**. GitHub's product documentation adds practical ones about task scope, environment setup and how feedback reaches the agent. Learn each anti-pattern as a symptom/mitigation pair.

Anti-pattern	What you see	Mitigation (prefer enforcement over advice)
Vague one-line issue ("Fix the login bug")	PR touches the wrong area, misses changes, or solves a different problem	Rewrite the issue as a prompt: problem, expected behaviour, acceptance criteria, files or directories to change; enforce with an issue form
Ambiguous implementation steps	Agent builds its plan on a guess	Resolve ambiguity *before* assignment: acceptance criteria plus directions; for uncertain work use a plan-first workflow
Over-broad scope (epic-sized task)	Huge diff, session times out (cloud agent sessions stop at 59 minutes), hard review	Split into focused issues; one task produces one branch and one PR; set a lower `timeout-minutes` if you want earlier failure
Planless execution	PR has a diff but no rationale	Require a plan section (PR template, plan file); use plan mode or a planner agent first
No tests and no runnable environment	Agent cannot validate; CI fails after push	Put build and test commands in `.github/copilot-instructions.md`; preinstall dependencies with `copilot-setup-steps.yml`; make CI a required check

Anti-pattern	What you see	Mitigation (prefer enforcement over advice)
Too many tools / over-permissioned agent	Planner edits files; agent calls unexpected MCP tools; larger blast radius	Explicit `tools` allowlist per custom agent; enable narrow MCP tools (`server/tool`) rather than `server/*` ; least-privilege tokens
Agent edits CI to make checks pass	Changes in `.github/workflows/` , skipped tests, relaxed thresholds	CODEOWNERS on `/.github/workflows/` plus required code owner review; keep "Require approval for workflow runs" on; instruct "fix the code, not the check"; optionally a `preToolUse` hook that denies edits to workflow files
Hidden reasoning	Only the final diff is visible	Session logs, PR description kept current, plan artifact, evidence links to workflow runs
Blind trust in automation	"CI passed, ship it"	Success criteria that reflect intent, CODEOWNERS, required reviews, risk-based approvals
Late feedback in the wrong place	Clarifications added as issue comments after assignment are ignored	Copilot receives the issue title, description and existing comments at assignment time only; post follow-ups on the PR with `@copilot`
Instructions treated as enforcement	"Do not edit files" in a prompt, yet edits happen	Treat instructions as guidance; use tool allowlists, plan mode, hooks, rulesets and required checks as enforcement
Dumping reference material into instructions	Bloated always-on context that goes stale	Keep instructions short and stable; expose large or external knowledge through an MCP server the agent calls on demand

COMMON TRAP

Answers such as "enable Copilot Memory", "use a larger runner" or "add more MCP servers" look attractive for a poor result from a vague issue. None of them creates missing requirements. If the root cause is an under-specified task, the fix is a better-scoped issue with acceptance criteria.

COMMON TRAP

Turning off **Require approval for workflow runs** to "speed things up" removes the guard that stops unreviewed agent code from running privileged workflows. GitHub's own warning says this may let unreviewed code gain write access to the repository or reach Actions secrets. Inspect changes to `.github/workflows/` before you click **Approve and run workflows**.

1.1.3 Define inputs, outputs, and success criteria for agents

The training module frames every agent task as a contract with three parts. **Inputs**: what the agent needs (issue context, constraints, boundaries). **Outputs**: what it produces (a plan, a PR, evidence). **Success criteria**: how results are judged (checks, scans, review outcomes). When tasks are under-specified, agents produce changes that *look* plausible but do not solve the underlying problem.

Contract element	Concrete GitHub mechanism	Notes
Task description	Issue title, body and existing comments; **Optional prompt** field when assigning	Comments added after assignment are not seen; use the PR for follow-ups
Starting point	Repository and base branch dropdowns at assignment or task start	If no branch is chosen, the default branch is used; choose `release/x.y` deterministically for hotfixes rather than asking in prose

Contract element	Concrete GitHub mechanism	Notes
Role and tools	Custom agent selected from the agent dropdown; model and reasoning level dropdowns	Agent profile versions follow the Git commit SHA of the profile file on that branch
Standing conventions	`.github/copilot-instructions.md`, `.github/instructions/**/*.instructions.md`, `AGENTS.md` (nearest file wins), `CLAUDE.md`, `GEMINI.md`; organization instructions	Repository-wide instructions also apply to Copilot Chat and Copilot code review
Environment	`.github/workflows/copilot-setup-steps.yml`; Agents secrets and variables; firewall allowlist	Setup steps run before the agent starts and must be on the default branch
External knowledge	MCP servers (repository settings or `mcp-servers` in an agent profile)	GitHub and Playwright MCP servers are available by default to the cloud agent
Output	Branch, commits, (draft) PR with an up-to-date description, session log	One branch and one PR per task, in one repository
Success criteria	Required status checks, CodeQL and dependency results, review outcomes, scope matches intent	"CI passed" is necessary but not always sufficient

An ideal issue for Copilot, according to GitHub's best-practice guide, has three things: a clear description of the problem or required work, complete acceptance criteria describing what a good solution looks like (for example, whether unit tests are required), and directions about which files need to change. The agent can search the codebase, including semantic code search, so exact paths are helpful rather than mandatory. An issue form makes these fields hard to skip.

Issue forms PREVIEW are YAML files in `.github/ISSUE_TEMPLATE/`. Each form field becomes Markdown in the issue body, which is exactly what the agent reads.

Listing 1.1: Issue form that captures an agent task contract (.github/ISSUE_TEMPLATE/agent-task.yml)

```yaml
name: Agent-ready task
description: A well-scoped task that can be assigned to Copilot cloud agent.
title: "[Agent]: "
labels: ["agent-ready"]
body:
  - type: markdown
    attributes:
      value: |
        This issue is the agent's prompt. Be specific about the problem,
        the boundaries, and how we will know the work is done.
  - type: textarea
    id: problem
    attributes:
      label: Problem and expected behaviour
      description: What is wrong or missing, and what should happen instead?
    validations:
      required: true
  - type: textarea
    id: scope
    attributes:
      label: In scope (files, directories, components)
      placeholder: "src/auth/session.ts, tests/auth/"
    validations:
      required: true
  - type: textarea
    id: out-of-scope
    attributes:
      label: Out of scope
      value: |
        - No changes under .github/workflows/ or infra/
        - No new runtime dependencies without discussion
```

```
  - type: textarea
    id: acceptance
    attributes:
      label: Acceptance criteria
      value: |
        - [ ] Expired sessions redirect to /login with a 401 logged once
        - [ ] Unit tests cover expired, valid and missing tokens
        - [ ] `npm test` and `npm run lint` pass
    validations:
      required: true
  - type: dropdown
    id: risk
    attributes:
      label: Risk level
      options:
        - Low (docs, formatting)
        - Medium (application code, dependency bumps)
        - High (auth, infra, workflows) - plan-first required
    validations:
      required: true
```

Repository custom instructions supply the conventions the agent should apply to *every* task: how to build, test and validate, the repository layout, and rules of engagement. GitHub's troubleshooting guide notes that if Copilot pushes changes that fail your CI, the best fix is clear instructions in `.github/copilot-instructions.md` on how to validate before pushing.

Listing 1.2: Repository-wide instructions (.github/copilot-instructions.md)

```
This is a TypeScript service (Node.js 20) that issues and validates user sessions.

## Development flow
- Install: `npm ci`
- Build: `npm run build`
- Test: `npm test` (Jest). Run the full suite before every push.
- Lint: `npm run lint`. Fix lint errors; do not disable rules inline.

## Repository structure
- `src/auth/`: session and token logic
- `src/api/`: HTTP handlers (thin; no business logic)
- `tests/`: Jest tests mirroring `src/`
- `.github/workflows/`: CI owned by the platform team

## Rules of engagement
1. Keep changes within the scope stated in the issue. List any extra file you had to touch and why.
2. Never modify files under `.github/workflows/` or `infra/` unless the issue explicitly asks for it.
3. If a check fails, fix the code or the test that is genuinely wrong. Never skip, delete or weaken a test or
a check to make it pass.
4. Add or update unit tests for every behaviour change.
5. In the pull request description, include: Plan, Scope, Validation performed (commands and results), Risks.
```

Path-specific instructions add rules only for matching files. The `applyTo` glob selects files, and the optional `excludeAgent` key (`"code-review"` or `"cloud-agent"`) stops one of the two agents from using the file.

Listing 1.3: Path-specific instructions for tests (.github/instructions/tests.instructions.md)

```
---
applyTo: "tests/**/*.test.ts"
---

- Use table-driven tests (`it.each`) for input variations.
- Each test must be independent; reset mocks in `afterEach`.
- Name tests after the behaviour, not the function: "rejects an expired token".
```

An agent that cannot install dependencies cannot validate its work. Copilot can discover and install dependencies by trial and error, but GitHub calls this slow and unreliable and sometimes impossible (for private packages). The fix is a Copilot setup steps workflow at `.github/workflows/copilot-setup-steps.yml`. It must contain a single job named `copilot-setup-steps`; only `steps`, `permissions`, `runs-on`, `services`, `snapshot` and `timeout-minutes` (maximum 59) can be customised, and the file only takes effect from the default branch. If a setup step fails, Copilot skips the remaining steps and starts work with whatever state the environment is in.

Listing 1.4: Copilot setup steps (.github/workflows/copilot-setup-steps.yml)

```yaml
name: "Copilot Setup Steps"

# Runs automatically when the file changes, and manually from the Actions tab
on:
  workflow_dispatch:
  push:
    paths:
      - .github/workflows/copilot-setup-steps.yml
  pull_request:
    paths:
      - .github/workflows/copilot-setup-steps.yml

jobs:
  # The job MUST be called copilot-setup-steps
  copilot-setup-steps:
    runs-on: ubuntu-latest
    timeout-minutes: 30          # fail earlier than the 59-minute ceiling
    permissions:
      contents: read             # lowest permission needed to clone
    steps:
      - name: Checkout code
        uses: actions/checkout@v6

      - name: Set up Node.js
        uses: actions/setup-node@v7
        with:
          node-version: "20"
          cache: "npm"

      - name: Install dependencies
        run: npm ci
```

PRODUCTION NOTE

Write success criteria that reflect the *intent* of the task, not only "tests pass". For a vulnerability fix, "the vulnerable version is no longer resolvable, including transitively" is the real criterion; a green build alone could hide a transitive copy of the vulnerable package. Then make the automatable part a **required status check** through a ruleset or branch protection so success is enforced by the system rather than asserted by the agent.

EXAM TIP

Know which file solves which problem: wrong scope or missing requirements means the **issue**; conventions and validation commands mean **custom instructions**; missing dependencies or tools in the agent environment mean `copilot-setup-steps.yml`; external or internal knowledge not in the repository means an **MCP server**; role-specific behaviour and tool limits mean a **custom agent profile**.

1.2 Define boundaries between planning, reasoning, and action

Reliable agent systems separate three things: **planning** (what will be done and why), **execution** (the concrete changes) and **validation** (evidence that the outcome meets the success criteria). When planning and execution are mixed, reviewers see only the final diff and lose the chance to validate intent early, catch misunderstandings and control scope before

impact. GitHub maps the three naturally: the plan lives in an issue comment, PR description or plan file; execution lives in commits on a branch; validation lives in checks, scans, artifacts and reviews.

1.2.1 Configure agent planning to be distinct from agent execution

The official training lists four ways to enforce a planning boundary. Know all four and which product feature implements each.

Boundary technique	How it is implemented
Capability boundary: planning agents are read-only	Custom agent with a `tools` allowlist that omits `edit` and `execute` (for example `search`, `read`, `web/fetch`, selected MCP tools)
Explicit transition (handoff) to an implementation agent	VS Code `handoffs` in the planner's front matter, with `send: false` so a human submits the prompt
Tool gating in orchestrators	Planner coordinates subagents restricted by the `agents` property; implementation tools enabled only in the implementer profile
Plan mode workflows	VS Code Plan mode (`/plan`), the Local harness built-in Plan agent, Copilot CLI plan mode (`/plan`, `Shift+Tab`, `--plan`)

VS Code. VS Code runs chat through an *agent harness* chosen from the **Session Target** control (for example Copilot, Local, Claude, or the Cloud target that delegates to Copilot cloud agent). In a **Copilot** session, **Plan** is a mode: type `/plan` followed by the outcome, constraints and how you will verify success, or select Plan in the mode picker. The agent researches the project, asks clarifying questions and presents a plan for review. In a **Local** session, **Plan** is a built-in agent role (alongside Ask and Agent) that researches without editing project files and stores its plan in session memory at `/memories/session/plan.md`. In both cases, VS Code documentation stresses that simply asking the current agent "do not change files" guides behaviour but *does not change the session's permissions*; selecting Plan does.

Custom agents. For a repeatable planning workflow with your own deliverables, define a planner custom agent. VS Code custom agents are `.agent.md` files in `.github/agents/` (workspace) or `~/.copilot/agents/` (user). The same profile format is read by Copilot cloud agent and Copilot CLI; unknown properties and unavailable tools are ignored, which keeps profiles portable. If `name` is omitted, the file name (without `.agent.md`) becomes the display name.

Listing 1.5: Read-only planner with a manual handoff (.github/agents/planner.agent.md)

```
---
name: planner
description: Researches the codebase and produces a structured implementation plan. Never edits files.
argument-hint: Describe the feature or change you want planned
tools: ['search', 'read', 'web/fetch', 'microsoftdocs/microsoft_docs_search']
agents: []
handoffs:
  - label: Start implementation
    agent: implementer
    prompt: Implement the approved plan above. Follow it step by step and stop if a step is unclear.
    send: false
---
# Planning instructions
You are a planning agent. Research the repository and produce a plan.
Do not propose code edits inline and do not modify any file.

Return the plan as Markdown with exactly these sections:

## Overview
## Requirements and acceptance criteria
## Affected files and existing patterns to reuse
## Implementation steps (numbered, each independently testable)
## Verification (tests to add, commands to run, manual checks)
## Risks, assumptions and open questions
## Out of scope
```

Points to notice in Listing 1.5: the tool list contains only read-oriented tools plus one MCP tool referenced as `server/tool` (the server must also be configured, for example in `.vscode/mcp.json`; listing a tool name does not install or authenticate a server); `agents: []` prevents subagent use; and the handoff uses `send: false` (the default), so clicking **Start implementation** only pre-fills the prompt for the implementer. A human still has to read the plan and press send.

Listing 1.6: Implementation agent that receives the handoff (.github/agents/implementer.agent.md)

```
---
name: implementer
description: Implements an approved plan with minimal, focused edits and runs the tests.
tools: ['read', 'search', 'edit', 'execute']
user-invocable: true
disable-model-invocation: true
handoffs:
  - label: Review changes
    agent: reviewer
    prompt: Review the changes made for the plan above against our conventions. Report findings only.
    send: false
---
Implement only the steps in the approved plan. Follow existing code patterns.
After each step, run `npm test`. If a step cannot be completed as written,
stop and explain why instead of improvising a different design.
```

`disable-model-invocation: true` prevents other agents from invoking the implementer automatically as a subagent, so it runs only when a person selects it or follows the handoff. `user-invocable: false` does the opposite: it hides an agent from the picker while keeping it available as a subagent.

Property	Effect	Where it applies
`tools` omitted or `["*"]`	All available tools, including configured MCP tools	All surfaces
`tools: []`	No tools at all	All surfaces
`tools: [read, search]`	Only listed tools; unknown names ignored	All surfaces
`handoffs`, `argument-hint`	Suggested next-step buttons; hint text	VS Code and IDEs; ignored by cloud agent on GitHub.com
`mcp-servers`, `metadata`	Agent-scoped MCP servers; annotations	GitHub.com cloud agent and CLI; not used in VS Code and other IDEs
`user-invocable: false`	Hidden from the agent picker; subagent only	All surfaces
`disable-model-invocation: true`	Cannot be chosen automatically by the model or invoked as a subagent	All surfaces (replaces retired `infer`)
`target`	`vscode` or `github-copilot`; unset means both	All surfaces

Copilot CLI. `/plan` starts a planning session where the CLI can explore and analyse your codebase but is blocked from editing project files. The documentation is precise about the guarantee: edits, patches and shell commands that would change workspace files are blocked by the tool layer rather than by asking the model nicely; the plan itself can still be written in a private planning workspace; helper sessions inherit the restriction. It is described as "a safety net, not a guarantee": shell commands whose effect cannot be determined in advance, and calls to external or MCP tools, may be allowed through. You approve the plan to leave plan mode.

Copilot cloud agent. On GitHub.com you can start a session from a prompt, ask research questions, ask Copilot to "create a plan", iterate until the plan matches your intent, and only then say "implement the plan we agreed upon". Sessions started this way do not create pull requests automatically.

COMMON TRAP

A planner with `agent` in its tool list can spawn subagents that may hold `edit` or `execute`, which quietly re-opens the write path. For a strictly read-only planner, omit `agent` or restrict `agents` to read-only helpers.

1.2.2 Configure an agent to output a structured plan

A plan is only reviewable if it has a predictable shape. Three mechanisms produce structure:

- **A plan template in the agent body.** Listing 1.5 dictates headings. GitHub's customization library sample *implementation-planner* does the same with Overview (including success criteria), Technical Approach, Implementation Plan by phases with Small/Medium/Large estimates and dependencies, Considerations (assumptions, constraints, risks) and Not Included.
- **A plan file.** VS Code Copilot Plan mode can expose the plan as a file you open with **Open Full Plan**, edit and comment inline; the Local Plan agent keeps `plan.md` in session memory and **Open in Editor** creates an editable copy you can save in the repository. In the CLI, the approved plan file sits in the planning workspace.
- **A PR-level plan.** A pull request template with a mandatory Plan section, so every agent PR carries the same plan and evidence headings.

There is a deliberate trade-off between a planner that *responds* with a plan and one that *writes* a plan file. GitHub's sample implementation-planner profile lists `tools: ["read", "search", "edit"]` because its instructions ask it to create Markdown plan files. If your requirement says the planner must not change files, drop `edit` and have it return the plan in its response (or rely on plan mode, which can write only to its own planning workspace).

Listing 1.7: PR template that requires a plan and evidence (.github/pull_request_template.md)

```
## Plan (required)
- **Goal / linked issue:**
- **Scope (paths):**
- **Steps:**
  1.
  2.
- **Success criteria (verifiable):**
  - [ ] Required checks pass
  - [ ] Acceptance criteria from the issue met
- **Risks and mitigations:**
- **Rollback / escalation:**

## Evidence
- Workflow run(s):
- Test report artifact:
- Agent session log:

## Review checklist
- [ ] Plan reviewed before code review
- [ ] Changes stay within stated scope
- [ ] CODEOWNERS reviews complete for sensitive paths
```

EXAM TIP

A good plan, according to the official training, states **scope** (what will change), **success criteria** (how you will know it worked) and a **rollback or escalation path**. If an answer option produces a plan with those three elements in an inspectable artifact, it is usually the right one.

1.2.3 Validate agent plans

Validation means checking the plan against intent before code exists. VS Code's planning guide gives a four-point review that is worth memorising:

Check	Question to ask
Scope	Does it solve the requested problem and exclude unrelated changes?
Project fit	Does it name the affected files and reuse existing patterns?
Decisions	Are assumptions, dependencies and unresolved choices explicit?
Verification	Do the proposed tests and manual checks prove the change works without breaking existing behaviour?

Tools that support validation:

- **VS Code Review Plan card. Open Full Plan** opens the plan file; you can edit it or add inline comments, then add overall feedback and click **Submit Feedback**. This sends comments to the agent *without* approving implementation. Pending feedback disables approval until you submit or clear it.
- **Clarifying questions.** Plan mode asks questions while it researches; answer them or ask it to recommend an option and explain trade-offs.
- **Second opinion.** In Copilot CLI, `/rubber-duck` consults an agent that uses a complementary model to critique plans, code and tests.
- **Human reviewers on GitHub.** In a plan-first PR, reviewers discuss and approve the plan in the PR; CODEOWNERS routes plans that touch sensitive paths to the right experts.
- **Mechanical gates.** A required check can fail a PR whose description lacks the Plan section. This turns "include a plan" from expectation into enforcement.

PRODUCTION NOTE

Use a strong reasoning model for planning and a cheaper model for implementing a reviewed, well-scoped plan. VS Code documents this pattern: choose **Approve Plan Only**, switch the model and the mode from Plan to Interactive, then ask the agent to implement the approved plan. Compare total cost, because retries and rework can outweigh savings.

1.2.4 Prevent agent action until the agent checked and approved

"Prevent action until approved" is about *gates*: points where execution cannot continue without a human decision. The exam expects you to pick the gate that actually blocks, not the one that merely asks.

Surface	Gate	What it blocks until a human acts
VS Code custom agents	Handoff with `send: false`	The implementation prompt is pre-filled but not submitted
VS Code Plan mode (Copilot harness)	Plan review card: **Implement Plan**, **Approve Plan Only**, **Implement with Autopilot**, **Reject**	Implementation starts only after you choose an action
VS Code Plan agent (Local harness)	**Start Implementation** button	Switches to Agent and submits the implementation request
VS Code tool calls	Manual permissions (default) and tool, URL and terminal approvals	Each non-auto-approved edit, command or external call
Copilot CLI	Plan mode; approve plan to exit	Edits to project files
Cloud agent on GitHub.com	Prompt-started session on a branch; **Create pull request** when ready	The PR is not created until you decide
Cloud agent PR	Draft PR; agent cannot mark ready, approve or merge	Merge
GitHub Actions	**Approve and run workflows** (default)	Workflow runs on agent pushes
Deployments	Environment with required reviewers	Jobs targeting that environment, and its secrets

Two settings deliberately *remove* a gate, and questions often hinge on recognising them:

- **Plan-then-autopilot** in Copilot CLI (`--plan --mode autopilot`, or the `COPILOT_PLAN_THEN_AUTOPILOT` environment variable) starts in plan mode and automatically continues into autopilot once the plan is ready, *without waiting for a human to approve the transition*.
- **Implement with Autopilot** in VS Code (and **Start with Autopilot** PREVIEW in the Local harness) approve the plan and continue autonomously with automatic tool approval. Approving a plan and approving tool actions are separate decisions; check the session's permission level before implementing.

Listing 1.8: Copilot CLI commands for plan-first work

```
# Start an interactive session directly in plan mode
copilot --plan

# Inside a session: create a plan before coding (edits to project files are blocked)
/plan Extract the retry logic in src/payments into a RetryPolicy class; keep the public API unchanged

# Get a second opinion on the plan from a complementary model
/rubber-duck Critique this plan for missing edge cases and test gaps

# Shift+Tab cycles standard, plan and autopilot modes in the interactive UI

# Review local changes, then run the code review agent
/diff
/review Focus on error handling and missing tests

# Hand the approved work to Copilot cloud agent (branch + draft PR on GitHub)
/delegate Implement the approved RetryPolicy plan and add unit tests

# Non-interactive run of a specific custom agent; deny rules win even over allow-all
copilot --agent=planner --allow-all-tools --deny-tool='write' \
  -p "Produce an implementation plan for the RetryPolicy refactor" \
  --share=./retry-plan-session.md

# AVOID when a human must approve the plan: continues to autopilot automatically
copilot --plan --mode autopilot
```

COMMON TRAP

`--agent` selects a custom agent but adds no approval phase; `--allow-all` (and its alias `--yolo`) grants all tool, path and URL permissions; `/compact` only summarises history to save context. None of these creates a review point before edits. Only plan mode (or a read-only agent plus a human-sent handoff) does.

NOTE

The training module describes two governance options for plan approval on GitHub. **Plan-first PR**: the agent opens a PR with only the plan; reviewers approve it; implementation follows in later commits or a new PR. Use it for high-risk or hard-to-reverse work (workflows, infrastructure, authentication, production). **Plan + execution in the same PR**: the plan sits in the description next to initial commits; faster, suitable for low or medium risk, with required checks and reviews still blocking merge. The choice is not *whether* work is reviewed but *when* code may be generated relative to human validation.

1.3 Configure observability and control for autonomous agents

Autonomy is not a binary switch. The training module puts it bluntly: *autonomy must be designed, not assumed*. You decide how far each class of task may proceed without a human, wrap that decision in guardrails that do not depend on the model's good behaviour, make every step inspectable in normal GitHub tooling, and place human intervention where it materially reduces risk.

1.3.1 Plan and implement the degree of agent autonomy, including guardrails

Start with a risk-based autonomy model. The official training uses paths as the risk signal, which maps cleanly onto CODEOWNERS and rulesets.

Risk	Example paths	Autonomy design
Low	`docs/`, formatting	Auto-merge after required checks (and reviews, if configured) pass
Medium	`src/`, dependency bumps	PR required, checks required, at least one review
High	`infra/`, `.github/workflows/`	CODEOWNERS, multiple reviews, stricter rulesets; plan-first
Critical	Production deployment settings, secrets	Environment approvals; the agent prepares but cannot execute

Next, choose the execution mode for each surface. The ladder below runs from least to most autonomous.

Level	VS Code	Copilot CLI	GitHub.com
Read-only	Ask role; Plan mode or Plan agent; read-only custom agent	Plan mode	Copilot Chat; research session
Supervised execution	Agent role with **Manual permissions** (default): non-auto-approved tools need confirmation	Standard (interactive) mode with permission prompts; `--allow-tool` / `--deny-tool` rules	Prompt-started cloud agent session on a branch, steered and reviewed before a PR
Assisted	**Assisted permissions**: an LLM judge assesses each tool call; unapproved calls need confirmation	n/a	n/a
Autonomous, local	**Allow all** permission level; **Autopilot** mode	Autopilot mode (`--autopilot`, optionally `--max-autopilot-continues`); `--allow-all` / `--yolo`	n/a
Autonomous, background	Cloud target (delegates to cloud agent)	`/delegate` or `&` prefix	Issue assigned to Copilot cloud agent; automations on schedule or events

Guardrails are the controls that hold regardless of what the model decides. Group them by layer:

Layer	Guardrail	Notes
Capability	Custom agent `tools` allowlist; narrow MCP tool selection	Treat changes to tool allowlists as governance-sensitive changes
Local permissions	VS Code `chat.permissions.default`, `chat.tools.eligibleForAutoApproval` (set a tool to `false` to always require approval), `chat.tools.terminal.autoApprove`, sandboxing (`chat.agent.sandbox.enabled`); CLI `--deny-tool` (deny always beats allow) and the managed `permissions.disableBypassPermissionsMode` setting that suppresses all `--allow-all*` options	Organisations can hide **Assisted permissions** or **Allow all** through enterprise-managed settings
Runtime	Cloud agent firewall (recommended allowlist on by default); Agents secrets and variables; least-privilege `permissions` in workflows	The firewall only covers processes the agent starts through its Bash tool, not MCP server processes or setup steps

Layer	Guardrail	Notes
Repository write path	Agent pushes only to its own branch; only users with write access can trigger it; subject to branch protections and required checks	Incompatible rules (for example commit-author restrictions) block the agent; rulesets can add Copilot as a bypass actor
Merge	Required reviews, CODEOWNERS, required checks, rulesets	Requester cannot approve; an extra approval is required for Copilot PRs not attributed to a person
Execution of CI	**Require approval for workflow runs** (on by default)	Repository administrators can disable it under **Settings > Copilot > Cloud agent**
Policy hooks	`.github/hooks/*.json` with `preToolUse` returning `permissionDecision: "deny"`	Supported by Copilot cloud agent and Copilot CLI; enforcement independent of model reasoning
Deployment	Environments with required reviewers	Jobs wait for approval before accessing protected secrets or deploying

Listing 1.9: CODEOWNERS that routes high-risk paths to owners (.github/CODEOWNERS)

```
# Combine with "Require review from Code Owners" in a ruleset or branch protection
*                    @contoso/core-team
/.github/workflows/  @contoso/platform-team
/infra/              @contoso/platform-team
/src/auth/           @contoso/security-team
/.github/agents/     @contoso/platform-team
/.github/hooks/      @contoso/platform-team
```

Hooks run custom shell commands at points in an agent session: `sessionStart`, `sessionEnd`, `userPromptSubmitted`, `preToolUse`, `postToolUse`, `agentStop`, `subagentStop` and `errorOccurred`. Only `preToolUse` can approve or deny a tool call. The file must contain `"version": 1` and a `hooks` object. On the cloud agent only the `bash` field is honoured, and because no user is present, a decision of `"ask"` is treated as `"deny"`. Command `preToolUse` hooks fail closed on errors and non-zero exits but fail open on timeouts, so keep them fast.

Listing 1.10: Hook that blocks agent edits to workflow files (.github/hooks/guardrails.json and script)

```
{
  "version": 1,
  "hooks": {
    "preToolUse": [
      {
        "type": "command",
        "bash": "./scripts/hooks/deny-workflow-edits.sh",
        "timeoutSec": 10
      }
    ],
    "postToolUse": [
      {
        "type": "command",
        "bash": "./scripts/hooks/log-tool-use.sh"
```

```
      }
    ]
  }
}

# scripts/hooks/deny-workflow-edits.sh
#!/usr/bin/env bash
input="$(cat)"   # JSON payload: sessionId, timestamp, cwd, toolName, toolArgs
if echo "$input" | grep -q '\.github/workflows/'; then
  echo '{"permissionDecision":"deny","permissionDecisionReason":"Workflow files are owned by the platform
team. Fix the code, not the pipeline."}'
  exit 0
fi
echo '{"permissionDecision":"allow"}'
```

EXAM TIP

When a scenario asks for a control that works "regardless of the prompt" or "even if the agent ignores instructions", look for a tool allowlist, plan mode, a `preToolUse` hook, a ruleset, a required check, CODEOWNERS with required review, or an environment approval. Text in `copilot-instructions.md` or in the agent body is guidance only.

PRODUCTION NOTE

Copilot automations run the cloud agent on a schedule or in response to events without a person initiating each task. GitHub mitigates the extra risk: work is attributed to the automation's creator (who therefore cannot approve its PRs), you choose exactly which tools an automation can use, events from users without write access are ignored by default, and Actions workflows on resulting PRs still need approval, which stops one automation from silently triggering a chain of others.

1.3.2 Configure agent to produce inspectable artifacts within standard development tooling

An agent system must leave a visible artifact for every meaningful action. The training module defines a **minimum observability set**: a visible plan artifact, a bounded PR and commit history, links to workflow runs for required checks, durable artifacts (logs, reports, traces) and review outcomes. A matching **minimum audit trail** is: stated goal, inspectable plan, bounded changeset, automated evidence, human judgment, clear outcome (merge, revert or escalation). The point is to make any later question answerable: what changed, who approved it, what evidence existed, and what happened next.

Artifact	Where you find it	What it proves
Eyes reaction on the issue or comment	Issue or PR comment	Copilot has picked up the task or comment
Draft pull request linked to the issue	Issue timeline, PR list	Work started on a `copilot/` branch
"Copilot started work" timeline event and **View session**	PR timeline	A session exists; logs stream live
PR title and body kept up to date	PR description	Current plan, progress and summary of changes
Session log and overview	Agents panel, agents page (`github.com/copilot/agents`), repository **Agents** tab	Reasoning, tools used, setup steps output, validation run, token usage, session length
Commits authored by Copilot, co-authored by the requester, signed ("Verified"), with a link to the session log in each message	Commit history	Who started the work and why each change was made
Firewall warning	PR body or comment	Blocked address and the command that tried to reach it

Artifact	Where you find it	What it proves
Workflow runs, checks and uploaded artifacts	Actions tab, PR checks	Objective validation evidence tied to a commit
Copilot code review comments	PR conversation and files	Automated review findings and suggested fixes
Audit log events	Organisation or enterprise audit log	Administrative traceability of agent activity
Plan files	VS Code plan file, CLI planning workspace, saved Markdown in the repository	Intent reviewed before execution
Shared session transcript	CLI `--share` (Markdown) or `--share-gist`; shared local sessions on the **Agents** tab	Reproducible record of a local or scripted run

Cloud agent sessions are shared by default and appear under "All sessions" on the repository's **Agents** tab for anyone with access. Local sessions from Copilot CLI, VS Code, JetBrains or the Copilot app are unshared by default and can be shared view-only. Cloud agent sessions can be archived but not deleted. Copilot Chat on GitHub.com can answer questions about a Copilot PR ("what changed, what was validated, and why") by drawing on the session logs.

Listing 1.11: Required CI check that publishes durable evidence (.github/workflows/ci.yml)

```
name: CI
on:
  pull_request:
    branches: [ main ]

permissions:
  contents: read

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: "20"
          cache: "npm"
      - run: npm ci
      - run: npm test -- --reporters=default --reporters=jest-junit
      - name: Upload test results
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: test-results-${{ github.sha }}
          path: junit.xml
          retention-days: 30
```

Naming the artifact after `github.sha` follows the training's traceability principle: evidence should be traceable to a specific workflow run and a specific commit. Mark the `test` job as a required status check in a ruleset so merge is blocked until it passes.

COMMON TRAP

Logs are not a substitute for review. GitHub explicitly says session logs show the work and tools used but "do not replace your own review and testing". Similarly, Copilot only responds to `@copilot` mentions on *open* pull requests; after merge or close, use a new issue or PR rather than a comment on the closed one.

1.3.3 Configure human intervention for autonomous agents without slowing delivery

The study guide wording matters: intervention must not slow delivery. The aim is to put humans at the few points where judgment changes the outcome, automate the rest, and keep feedback cheap. Use this toolbox:

Intervention	When to use it	Why it keeps delivery fast
Steer a running cloud agent session from the agents page prompt box	Agent is heading the wrong way mid-task	No restart; Copilot applies the input after its current tool call (each steering message consumes AI credits)
Stop session	Task no longer needed or prompt fundamentally wrong	Ends the Actions run and preserves commits already pushed
`@copilot` comments on the PR	Changes needed after the PR exists	Copilot pushes commits to the same branch; context from previous sessions on the PR is remembered; a PR created by a custom agent continues with that agent
Start a review instead of Add single comment	Several pieces of feedback	Copilot starts on each submitted comment; batching triggers one session for the whole review
Fix with Copilot on review comments (single or batch), on merge conflicts, or on a failing Actions job	Feedback already written by humans or Copilot code review	Delegates the fix with one click; option to commit directly or open a new PR
Automatic Copilot code review via ruleset (Automatically request Copilot code review, optionally Review new pushes and Review draft pull requests)	Every PR should get a first-pass review	Humans review a PR whose obvious issues are already flagged; a user setting and a ruleset are evaluated independently but produce only one review
Required reviews and CODEOWNERS	Merge accountability; sensitive paths	Only owners of risky paths are pulled in; low-risk paths stay fast
Approve and run workflows	Before CI runs on agent pushes	One click after inspecting workflow-file changes; keep it rather than disabling it
Escalation policy	Same required check fails repeatedly	Training guidance: after the same check fails twice, escalate with what failed, what was attempted, evidence and suggested next step; prevents infinite loops

Rules about who can intervene are testable details. Copilot only responds to comments from people with write access; comments from others are never presented to the agent. Your approval of a Copilot PR that you requested does not count toward required approvals, so another reviewer must approve. By default, Copilot code review leaves a review that does not count toward required approvals; **Copilot approvals** PREVIEW, when enabled in repository, organization and enterprise settings, can submit an approving review that satisfies the rule, and new pushes dismiss that approval.

EXAM TIP

"Minimise approvals that do not materially reduce risk" is a principle from the guardrails domain that also applies here. A good answer adds a human gate on high-risk paths (CODEOWNERS, environments) and automates low-risk paths (auto-merge after checks), rather than adding a manual approval to everything.

PRODUCTION NOTE

When you need to change the original requirement after assigning an issue, do not edit the issue and wait: the agent will not see it. Comment on the pull request with `@copilot` and the change, or steer the session from the agents page. If a session appears stuck it times out after an hour; you can retry by unassigning and reassigning the issue, or by repeating the comment.

Copilot cloud agent	Autonomous agent (formerly Copilot coding agent) that researches, plans and changes code in an ephemeral GitHub Actions environment, pushes to a `copilot/` branch and opens pull requests for human review.
System of record / control plane	GitHub stores the artifacts of agent work (issues, commits, PRs, runs, reviews) and enforces policy on it (reviews, checks, CODEOWNERS, rulesets, environments).
Task contract	The inputs, outputs and success criteria that define an agent task so outcomes are predictable and reviewable.
Agent profile	A Markdown file with YAML front matter (`.agent.md`) that defines a custom agent's description, instructions, tools, model, MCP servers and handoffs.
Handoff	A VS Code custom agent front-matter entry that shows a button to switch to another agent with a pre-filled prompt; `send: false` (default) requires a human to submit.
Plan mode	A planning workflow (VS Code Copilot harness, Copilot CLI) in which the agent researches and drafts a plan while edits to project files are blocked until the plan is approved.
Plan-first PR	Governance pattern where the agent's PR initially contains only a plan; implementation follows after human approval.
Copilot setup steps	The `.github/workflows/copilot-setup-steps.yml` workflow with a single `copilot-setup-steps` job that prepares the cloud agent environment before work begins.
Steering	Sending a follow-up prompt to a running cloud agent session so it changes direction without being stopped.
Session log	The record of an agent session's reasoning, tool calls and validation, linked from each agent commit and viewable from the agents panel or the PR.
Guardrail	A control that constrains agent behaviour independently of the model's reasoning, such as tool allowlists, hooks, rulesets, required checks and environment approvals.
Contributor model	Evaluating agent PRs with the same standards as human contributions: intent, scope, evidence, ownership, policy and fallback.

1.4 Additional practice questions

Practice 1.1 MULTIPLE CHOICE

Fabrikam assigns issue #412 to Copilot cloud agent. Ten minutes later the product owner adds a comment to the issue that tightens one acceptance criterion. The resulting pull request ignores the new criterion. You need Copilot to apply the change with the least delay. What should you do?

A. Edit `.github/copilot-instructions.md` to include the new criterion.
B. Add a comment on the pull request that mentions `@copilot` and states the new criterion.
C. Re-run the failed GitHub Actions workflow on the pull request.
D. Add the `agent-ready` label to the issue again.

Practice 1.2 MULTIPLE RESPONSE

You are creating a VS Code custom agent named `architect` that drafts implementation plans. A developer must review each plan before any file is changed, and the implementation must run in a separate agent named `builder`. Which two configurations should you include in `architect.agent.md`? Each correct answer presents part of the solution.

A. `tools: ['search', 'read', 'web/fetch']`
B. A `handoffs` entry with `agent: builder` and `send: false`
C. `tools: ['*']` and a body instruction "Never edit files"
D. `user-invocable: false`
E. A `handoffs` entry with `agent: builder` and `send: true`

Practice 1.3 YES/NO

Your repository requires one approving review on `main`. Dana assigns an issue to Copilot cloud agent, and Copilot opens a pull request. Default cloud agent settings apply. For each statement, select Yes if the statement is true. Otherwise, select No.

A. Copilot cloud agent can mark the pull request as ready for review and merge it after all checks pass.
B. GitHub Actions workflows do not run on Copilot's pushes until a user with write access selects **Approve and run workflows**.
C. Dana's approval satisfies the required approving review for this pull request.

Practice 1.4 MULTIPLE CHOICE

Team policy states: "For multi-file refactors in Copilot CLI, a human must approve the plan before any project file is edited." A developer proposes several ways to start sessions. Which command violates the policy?

A. `copilot --plan`
B. `copilot`, then `/plan Refactor the billing module`
C. `copilot --plan --mode autopilot`
D. `copilot`, then press `Shift+Tab` until plan mode is shown

Practice 1.5 MULTIPLE CHOICE

Copilot cloud agent sessions for a large migration issue repeatedly stop before completing, and the session logs show the sessions ending at the time limit. You need the work to complete reliably. What should you do?

A. Set `timeout-minutes: 120` in `copilot-setup-steps.yml`.
B. Move the agent to a larger GitHub-hosted runner.
C. Split the migration into several smaller, focused issues and assign them separately.
D. Disable the cloud agent firewall so dependencies download faster.

Practice 1.6 MULTIPLE RESPONSE

You assign an issue to Copilot cloud agent and type extra guidance in the **Optional prompt** field. Which information does Copilot use for this task? Select all that apply.

A. The issue title and description
B. Comments that already exist on the issue at assignment time
C. The text in the **Optional prompt** field
D. Comments added to the issue after assignment
E. The repository's `.github/copilot-instructions.md` file

Practice 1.7 MULTIPLE CHOICE

During review you notice that an agent-authored pull request changed `.github/workflows/ci.yml` to skip a failing test suite. You need to ensure that any future change to workflow files cannot be merged without approval from `@contoso/platform-team`, regardless of what the agent is told. What should you implement?

A. Add "Never modify workflow files" to `.github/copilot-instructions.md`.
B. Add a CODEOWNERS entry for `/.github/workflows/` and require review from code owners in a ruleset for `main`.
C. Enable automatic Copilot code review for all pull requests.
D. Disable the **Require approval for workflow runs** setting so that CI always runs.

Practice 1.8 MULTIPLE CHOICE

An auditor asks why a specific line in a merged commit authored by Copilot cloud agent was changed four months ago. You need the fastest route to the agent's reasoning for that commit. What should you use?

A. The link to the session logs included in the commit message
B. The workflow artifacts attached to the pull request's CI runs
C. Copilot Memory for the repository
D. The `copilot-setup-steps.yml` run history in the Actions tab

Practice 1.9 YES/NO

You use custom agents and planning in VS Code. For each statement, select Yes if the statement is true. Otherwise, select No.

A. When a handoff has `send: false`, selecting the handoff button switches to the target agent and pre-fills the prompt without submitting it.
B. In a Local harness session, the built-in Plan agent stores its plan in session memory at `/memories/session/plan.md`.
C. Writing "Do not change project files" in a prompt to the Agent role changes the session's permissions to read-only.

Practice 1.10 MULTIPLE CHOICE

Contoso wants every pull request in all repositories of its GitHub organization, including pull requests created by Copilot cloud agent, to receive an automated Copilot code review before human reviewers look at it, and to be re-reviewed after each new push. Developers must not need to change any personal settings. What should you configure?

A. A `.github/copilot-instructions.md` file in each repository that asks Copilot to review its own work
B. An organization ruleset with **Automatically request Copilot code review** and **Review new pushes** enabled, targeting all repositories
C. A `copilot-code-review.yml` workflow in each repository
D. A required status check that runs `copilot /review` on every pull request

Answers and explanations

1.1: B. Copilot receives the issue title, description and existing comments at assignment time and does not see later issue comments. Post follow-ups on the pull request with `@copilot`; Copilot pushes new commits to the same branch. Editing instructions (A) affects future tasks generally and is the wrong place for a task-specific criterion; re-running CI (C) and relabelling (D) give the agent no new information.

1.2: A and B. A read-only tool list is a capability boundary that enforces "no edits during planning". A handoff with `send: false` pre-fills the implementation prompt for `builder` but waits for the developer to submit it, which creates the review point. C relies on an instruction, which is guidance, not enforcement. D hides the agent from the picker. E auto-submits the prompt and

removes the human checkpoint.

1.3: No, Yes, No. The cloud agent cannot mark its PR ready for review, approve it or merge it. By default, Actions workflows wait for **Approve and run workflows** from a user with write access. The person who asked Copilot to create the PR cannot approve it, so another reviewer must provide the required approval.

1.4: C. `--plan --mode autopilot` enables plan-then-autopilot: the session moves from plan mode into autopilot automatically once the plan is ready, without waiting for a human to approve the transition. A, B and D all keep the session in plan mode until the user approves the plan.

1.5: C. Each cloud agent session has a maximum execution time of 59 minutes that cannot be extended; GitHub recommends breaking complex tasks into smaller, focused tasks. `timeout-minutes` in setup steps accepts at most 59 and can only shorten the limit (A). A larger runner (B) may speed builds but does not lift the ceiling or reduce scope. Disabling the firewall (D) weakens security and does not address the cause.

1.6: A, B, C and E. When you assign an issue, Copilot is sent the title, description, existing comments and any additional instructions you supply, and it also uses the repository's custom instructions. Comments added after assignment (D) are not seen.

1.7: B. CODEOWNERS routes changes under `/.github/workflows/` to the platform team, and a ruleset that requires code owner review makes their approval a merge requirement no matter what the agent does. A is guidance only. C adds review comments but does not enforce platform-team approval. D removes a safeguard and lets unreviewed workflow changes run.

1.8: A. Each commit authored by Copilot cloud agent includes a link to the agent session logs, which record its reasoning and tool use for review and audit. Artifacts (B) contain run outputs such as test reports and may have expired. Copilot Memory (C) stores learned repository facts, not per-commit reasoning. Setup-step runs (D) show environment preparation only.

1.9: Yes, Yes, No. With `send: false`, the handoff switches agent and pre-fills the prompt; the user submits it. The Local Plan agent keeps `plan.md` in session memory at `/memories/session/plan.md`. A prompt instruction guides the agent but does not change session permissions; select Plan mode or use a read-only agent for that.

1.10: B. Organization owners can configure automatic Copilot code review with a ruleset (branch rule **Automatically request Copilot code review**) that targets some or all repositories, and enable **Review new pushes**. This works independently of personal settings. Instructions (A) do not trigger reviews. `copilot-code-review.yml` (C) customises the code review environment, not when reviews happen. D invents a mechanism that is not how Copilot code review is triggered.

Key takeaways

- Map each agent to an SDLC stage with a GitHub artifact: plans in issues, PR descriptions or plan files; execution as commits on a branch; validation as checks, scans and reviews; deployment behind environment approvals.
- Copilot cloud agent turns issues into draft PRs in the background; Copilot code review adds automated review on PRs; agent mode in VS Code and Copilot CLI give interactive local control; `/delegate` moves CLI work to the cloud agent.
- The issue is the prompt: clear problem, acceptance criteria, files to change. Comments added after assignment are not seen; use `@copilot` on the PR.
- Inputs live in specific files: custom instructions (`.github/copilot-instructions.md` , `*.instructions.md` with `applyTo` , `AGENTS.md`), environment in `copilot-setup-steps.yml` (single job, default branch, `timeout-minutes` at most 59), knowledge via MCP, role via `.agent.md` .
- Separate planning from execution with enforcement, not wording: read-only `tools` , plan mode, handoffs with `send: false` , and a plan-first PR for high-risk work.
- Structured plans state scope, success criteria and a rollback or escalation path; validate them for scope, project fit, decisions and verification before approving.
- Know which options remove the human gate: `--plan --mode autopilot` , `--allow-all` / `--yolo` , **Allow all**, **Autopilot**, and disabling **Require approval for workflow runs**.
- Design autonomy by risk (low: auto-merge after checks; critical: agent prepares, cannot execute) and back it with guardrails: allowlists, firewall, hooks, CODEOWNERS, rulesets, environments.
- Inspectable artifacts include draft PRs, the "Copilot started work" event, live session logs, signed co-authored commits linking to session logs, Actions runs and artifacts, and audit log events.
- Intervene cheaply: steer sessions, batch PR feedback with **Start a review**, use **Fix with Copilot**, automate first-pass review, and escalate after repeated check failures.

Microsoft Learn references

- Study guide for Exam GH-600: Developing in Agentic AI Systems
- Foundations of Agentic AI in GitHub (training module)
- Explain the agent lifecycle: plan, act, evaluate
- Describe GitHub as the system of record and control plane
- Identify responsibilities, risks, anti-patterns, and traceability needs
- Apply the contributor model to agent-generated work
- Designing Agent Architecture and SDLC Integration (training module)
- Map agent responsibilities to the SDLC
- Define inputs, outputs, and success criteria
- Separate planning, reasoning, and execution
- Implementing PR governance with templates, checks, CODEOWNERS, rules, and environment gates
- Build reliable workflows: outputs, contexts, triggers, and cross-job handoffs
- Control and operate agents: observability, tools, MCP, secrets, hooks, and reliability
- Microsoft Learn MCP Server developer reference
- GitHub Docs: GitHub Copilot on GitHub.com
- GitHub Docs: Best practices for using GitHub Copilot to work on tasks
- GitHub Docs: Best practices for using GitHub Copilot
- GitHub Docs: Kick off a task with Copilot agents on GitHub
- GitHub Docs: Using Copilot cloud agent on GitHub
- GitHub Docs: Research, plan, and iterate on code changes with Copilot cloud agent
- GitHub Docs: Managing agent sessions
- GitHub Docs: Review output from Copilot
- GitHub Docs: Configuring settings for GitHub Copilot cloud agent
- GitHub Docs: Troubleshooting GitHub Copilot cloud agent
- GitHub Docs: Risks and mitigations for GitHub Copilot cloud agent
- GitHub Docs: Configure the development environment (copilot-setup-steps.yml)

- GitHub Docs: Customizing or disabling the firewall for GitHub Copilot
- GitHub Docs: Adding repository custom instructions for GitHub Copilot
- GitHub Docs: Support for different types of custom instructions
- GitHub Docs: About custom agents
- GitHub Docs: Creating custom agents for Copilot cloud agent
- GitHub Docs: Custom agents configuration
- GitHub Docs: Implementation planner (customization library)
- GitHub Docs: Preparing to use custom agents in your organization
- GitHub Docs: GitHub Copilot code review
- GitHub Docs: Configuring code review by GitHub Copilot
- GitHub Docs: GitHub Copilot CLI command reference
- GitHub Docs: Delegating tasks to Copilot (Copilot CLI)
- GitHub Docs: About hooks for GitHub Copilot
- GitHub Docs: GitHub Copilot hooks reference
- GitHub Docs: Syntax for issue forms
- VS Code: Custom agents in VS Code
- VS Code: Plan work with agents in VS Code
- VS Code: Manage approvals and permissions
- VS Code: Agent harnesses
- VS Code: Tools and context reference
- VS Code: MCP configuration reference
- VS Code release notes v1.126 (Edit mode deprecation)

DOMAIN 2

Implement tool use and environment interaction

Exam weight: 20–25%

An agent is only as capable, and only as dangerous, as the tools it can call and the environment it runs in. Domain 2 tests whether you can give an agent exactly the tools a task needs, wire in Model Context Protocol (MCP) servers safely, choose the right execution context (your editor, a terminal, a CI job or the Copilot cloud agent), and keep every autonomous action recoverable and attributable. It is the largest domain on GH-600, and most questions are configuration questions: which file, which key, which prefix, which flag, which setting page.

Study it in four passes that mirror the official outline. First, learn the **tool surfaces**: the `tools` property in an agent profile, tool aliases, MCP tool naming, and the approval models of VS Code and Copilot CLI. Second, learn the **MCP configuration formats**: VS Code `mcp.json`, the portable `.mcp.json`, the Copilot cloud agent repository JSON, and the `mcp-servers` block in a custom agent, then the remote GitHub MCP server and the organisation controls (registries and allowlists). Third, learn the **execution contexts** and their boundaries: repositories, `copilot/` branches, `copilot-setup-steps.yml`, the firewall, runners and Agents secrets. Fourth, learn the **safety net**: how errors surface, how to retry, how to roll back, how to escalate to a human and how to prove afterwards who did what.

Throughout, remember the product naming used by GitHub Docs today: **Copilot cloud agent** (formerly Copilot coding agent) is the autonomous agent that works in a GitHub Actions-powered environment and opens pull requests; **Copilot CLI** is the terminal agent (`copilot`); custom agents are Markdown agent profiles (`*.agent.md`) stored in `.github/agents/`.

Official skill (Domain 2)	Sub-skill	Covered in
Select and configure agent tools	Identify required tools	2.1.1
	Configure agent tools	2.1.2
	Configure tool permissions	2.1.3
Configure MCP servers	Add an MCP server as a tool to an agent	2.2.1
	Configure the GitHub remote MCP server	2.2.2
	Configure MCP registries	2.2.3
	Configure MCP allow lists	2.2.4
Integrate agents within development environments	Evaluate execution context	2.3.1
	Scope an agent to a specific repository	2.3.2
	Invoke an agent in a CI workflow	2.3.3
	Configure branch-based scope	2.3.4
	Enable autonomous actions, including creating branches and pull requests	2.3.5
	Apply environment-specific constraints	2.3.6

Official skill (Domain 2)	Sub-skill	Covered in
Operate agents with safe execution paths and robust error handling	Error handling	2.4.1
	Retries	2.4.2
	Rollbacks	2.4.3
	Escalation paths	2.4.4
	Traceability and accountability	2.4.5

2.1 Select and configure agent tools

A **tool** is a callable capability that the model can choose to invoke: read a file, search the workspace, apply an edit, run a shell command, fetch a URL, delegate to another agent, or call an operation exposed by an MCP server or an editor extension. The model decides *whether* to call a tool; you decide *which* tools exist for that agent and *whether* each call needs a human approval. Those are two separate layers, and the exam repeatedly checks that you do not confuse them.



2.1.1 Identify required tools

Start from the task, not from the tool catalogue. Break the task into verbs (inspect, change, validate, communicate) and map each verb to the narrowest capability that achieves it. GitHub's own guidance for MCP tools on the cloud agent is explicit: because the agent uses configured MCP tools autonomously, "we strongly recommend that you allowlist specific read-only tools". The same principle of least privilege applies to every surface.

Task verb	Typical capability	Agent profile alias / tool	Risk note
Understand code, answer questions, plan	Read files, search text and symbols	`read`, `search`	Low; still exposes repository content to the model
Change code or docs	Create and edit files	`edit`	Medium; review the diff before merge
Build, test, lint	Run shell commands	`execute`	High; a shell can do anything the account can do
Research external documentation	Fetch URLs, web search	`web` (not applicable to the cloud agent)	Prompt injection and data exfiltration
Split work across specialists	Invoke another custom agent / subagent	`agent`	The child agent brings its own tools
Read issues, pull requests, workflow runs	GitHub MCP server	`github/*` or `github/<tool>`	Token scope decides reach, not the prompt
Exercise a running web app	Playwright MCP server or browser tools	`playwright/*`	On the cloud agent it can only reach `localhost` by default
Query an external system (Sentry, Azure DevOps, Notion)	Third-party MCP server	`<server>/<tool>`	Needs a secret; prefer read-only tools

Three facts help you decide what is already available before you add anything:

- **Copilot cloud agent ships with two MCP servers enabled by default**: `github` and `playwright`. The GitHub MCP server uses "a specially scoped token that only has read-only access to the current repository"; the Playwright server "is only able to access web resources hosted within Copilot's own environment, accessible on `localhost` or `127.0.0.1`".
- **Copilot CLI ships built-in MCP servers** (`github-mcp-server`, `playwright`, `fetch`, `time`) that can be disabled with `--disable-builtin-mcps` or individually with `--disable-mcp-server SERVER-NAME`.
- **VS Code** exposes built-in tools grouped into tool sets such as `#read`, `#search`, `#edit`, `#execute`, `#web`, `#agent` and `#browser`, plus anything contributed by MCP servers and extensions. A single chat request can have at most **128 tools** enabled; beyond that you must deselect tools or enable virtual tools (`github.copilot.chat.virtualTools.threshold`).

> **EXAM TIP**
> When a scenario says "the agent must only read" or "the agent must not modify files", the answer is almost always a tool list such as `["read", "search"]`, not an instruction in the prompt body. Natural-language instructions shape behaviour; they do not grant or remove capabilities.

Also identify what the task does *not* need. A documentation agent needs `read`, `search` and `edit` but not `execute`; a reviewer needs `read` and `search` only; a test-fixing agent needs `execute` to run the test suite. More tools also cost context and decision quality: GitHub notes that enabling third-party MCP servers "may impact the performance of the agent and the quality of the outputs".

2.1.2 Configure agent tools

The `tools` property in an agent profile

Custom agents are Markdown files with YAML front matter. GitHub reads them from `.github/agents/` in a repository (file names such as `test-specialist.agent.md`), from `agents/` in an organisation's `.github` or `.github-private` repository, and, for enterprises, from the `.github-private` repository of a designated organisation. Copilot CLI also reads user-level profiles from `~/.copilot/agents/`. When names clash, "the lowest level configuration overrides higher-level configurations": repository beats organisation, organisation beats enterprise.

The `tools` property is a list of tool names or aliases (a YAML array or a comma-separated string). Its three states are the most testable fact in this objective:

Front matter	Effect
`tools` **omitted**	All available tools are enabled, including all MCP tools configured in the profile and in repository settings
`tools: ["*"]`	Same as omitted: all available tools
`tools: ["read", "edit", "search"]`	Only those tools (or aliases) are enabled
`tools: ["github/*", "sentry/get_issue"]`	All tools of one MCP server, plus one named tool of another
`tools: []`	All tools are disabled

Unrecognised tool names are ignored "which allows product-specific tools to be specified in an agent profile without causing problems". That is why one profile can list VS Code-only tools and still work on GitHub.com.

Tool aliases

Aliases let one profile work across VS Code, Copilot CLI and the cloud agent. All aliases are case-insensitive.

Primary alias	Compatible aliases	Cloud agent mapping	Purpose
`execute`	`shell`, `Bash`, `powershell`	`bash` or `powershell`	Run a command in the OS shell
`read`	`Read`, `NotebookRead`	`view`	Read file contents
`edit`	`Edit`, `MultiEdit`, `Write`, `NotebookEdit`	Edit tools such as `str_replace`	Edit files
`search`	`Grep`, `Glob`	`search`	Search files or text
`agent`	`custom-agent`, `Task`	Custom agent tools	Invoke a different custom agent
`web`	`WebSearch`, `WebFetch`	Currently not applicable	Fetch URLs and search the web
`todo`	`TodoWrite`	Currently not applicable	Structured task lists (supported in VS Code)

MCP tool naming: `server/tool`

In an agent profile, MCP tools are namespaced by the server name followed by a slash: `some-mcp-server/some-tool` enables one tool, and `some-mcp-server/*` enables every tool from that server. Extension tools in VS Code use the extension name as a proxy, for example `azure.some-extension/some-tool`. The server name is the key you gave the server in the MCP configuration (for example `sentry` in `"mcpServers": { "sentry": {...} }`), not the package name.

Do not confuse this with other naming schemes you may meet:

- **VS Code tool references** in chat and prompt bodies use `#`, for example `#web/fetch` or the tool set `#search`; in an agent body use `#tool:web/fetch`.
- **Copilot CLI permission patterns** use parentheses: `MyMCP(create_issue)` or just `MyMCP` for every tool of that server (see 2.1.3).
- **Copilot CLI internal tool names** sent to the model are sanitised as `serverName-toolName` (capped at 64 characters). You rarely type these.

Listing 2.1: Cloud-agent custom agent profile with scoped built-in and MCP tools (`.github/agents/dependency-auditor.agent.md`)

```
---
name: dependency-auditor
description: Audits dependency changes and reports risky upgrades without touching production code
target: github-copilot
tools: ["read", "search", "edit", "github/*", "sentry/get_issue_details"]
disable-model-invocation: true
---

You review dependency manifests and lock files in this repository.

- Use the GitHub MCP tools to read related issues and pull requests.
- Use Sentry only to look up issues referenced in the task.
- Write findings to docs/dependency-audit.md. Do not edit source files.
```

In Listing 2.1 the agent can read, search and edit files, call every GitHub MCP tool, and call exactly one Sentry tool, provided a server named `sentry` exists in the repository MCP configuration or in the profile's own `mcp-servers` block. It has no shell. `disable-model-invocation: true` means the cloud agent will not select it automatically from task context; a user must pick it. The Markdown body (up to 30,000 characters) carries behavioural instructions. The Sentry tool name is illustrative: take real tool names from the server's documentation, or from the **Start MCP Servers** step of a cloud agent session log, which lists the tools each server exposed.

Other profile properties that affect tool use

Property	Meaning	Surface notes
`description`	Required. Purpose and capabilities	All
`target`	`vscode` or `github-copilot`; unset means both	All
`model`	Model to use; unset inherits the default	VS Code also accepts a prioritised array
`disable-model-invocation`	Default `false`. On GitHub, `true` stops the cloud agent from auto-selecting the agent; in VS Code, `true` prevents invocation as a subagent	Replaces the retired `infer` (`infer: false` equals `disable-model-invocation: true`)
`user-invocable`	Default `true`. `false` hides the agent from the picker so it is only reachable programmatically or as a subagent	All
`mcp-servers`	Additional MCP servers for this agent only	Not used in VS Code and other IDE custom agents
`agents`	Which agents may be used as subagents; requires the `agent` tool	VS Code
`handoffs`, `argument-hint`	Suggested next steps; input hint	VS Code; ignored by the cloud agent

Listing 2.2: Review-only subagent hidden from the picker (VS Code and GitHub)

```
---
name: security-reviewer
description: Reviews changed files for security issues and reports findings only
tools: ["read", "search"]
user-invocable: false
---

Inspect the changed files you are given. Report findings as a numbered list with
file paths and line references. Never propose to run commands.
```

Listing 2.2 is the canonical "can look but cannot touch" pattern. Because `user-invocable` is `false`, it does not appear in the agents dropdown, but a coordinating agent that has the `agent` tool can still call it. Setting `disable-model-invocation: true` here would be wrong: in VS Code that blocks subagent use, which is the only way this agent is meant to run.

Tool sets in VS Code

A **tool set** is a named, reusable group of tools that you can reference in prompts, prompt files and custom agents as a single entity. VS Code provides predefined sets (for example `#edit` and `#search`); you create your own with **Chat: Configure Tool Sets**, which opens a `.jsonc` file. Each set has `tools`, `description` and `icon`.

Listing 2.3: A user-defined VS Code tool set

```
{
  "reader": {
    "tools": ["search/changes", "search/codebase", "read/problems", "search/usages"],
    "description": "Tools for reading and gathering context",
    "icon": "book"
  }
}
```

Reference it as `#reader` in chat or list `reader` in an agent's `tools`. If a listed tool is unavailable when the agent runs, VS Code ignores it.

Restricting tools in Copilot CLI and the SDK

Copilot CLI has a separate availability layer from its permission layer. `--available-tools` disables every tool except those listed; `--excluded-tools` removes only the listed tools. If you pass both, the allowlist in `--available-tools` wins and `--excluded-tools` is ignored. A tool outside the available set cannot be used "even if you specify it with the `--allow-tool` option". Valid values are the CLI's internal names such as `bash`, `view`, `edit`, `create`, `grep`, `glob`, `task` and `web_fetch`.

The Copilot SDK exposes the same idea programmatically: each entry in `customAgents` has a `tools` array (for example `["grep", "glob", "view"]` for a read-only explorer), where `null` or omitted means all tools, plus an optional per-agent `mcpServers` object.

> **COMMON TRAP**
> `tools: []` does not mean "no restriction"; it disables every tool. To grant everything, delete the `tools` line or use `["*"]`. Also, adding an `mcp-servers` block does not lift an existing restriction: if `tools` lists only `read` and `search`, the new server's tools stay unavailable until you add `server/*` or specific `server/tool` entries.

On the cloud agent, MCP configurations are merged in a fixed order: out-of-the-box servers (such as GitHub) first, then the custom agent's `mcp-servers`, then the repository MCP settings, "allowing each level to override settings from the previous level". The `tools` list then filters the merged result.

2.1.3 Configure tool permissions

Availability says what *could* be called; permission says whether a call runs without a human. Each surface implements permission differently.

Surface	Default behaviour	How to loosen	How to tighten
VS Code (Copilot harness)	**Manual permissions**: tools, URLs and terminal commands not auto-approved need confirmation	Per-tool approvals, `chat.tools.terminal.autoApprove`, **Assisted permissions**, **Allow all**, Autopilot mode	`chat.tools.eligibleForAutoApproval` set to `false`, sandboxing, enterprise-managed settings
Copilot CLI	Read-only operations run automatically; modifying tools, URLs and destructive commands prompt	`--allow-tool`, `--allow-url`, `--allow-all-tools`, `--allow-all` / `--yolo`	`--deny-tool`, `--deny-url`, `--available-tools`, managed `permissions.disableBypassPermissionsMode`
Copilot cloud agent	No interactive approvals; configured MCP tools are used "autonomously" and the agent "will not ask for your approval"	Add tools to the MCP `tools` array	List only specific read-only tools; restrict the agent profile `tools`; firewall

VS Code approvals and auto-approve

VS Code uses two layers: **approvals** decide whether an action runs automatically; **sandboxing** (Preview on macOS, Linux and WSL2) restricts file system and network access of terminal commands even after approval. New sessions use `chat.permissions.default`. The permission levels are **Manual permissions** (default), **Assisted permissions** (an LLM judge assesses each call; enable `chat.assistedPermissions.enabled`) and **Allow all**. **Autopilot** is a mode for Copilot harness sessions that auto-approves all tools, retries on errors and answers blocking questions automatically.

- **Chat: Manage Tool Approval** lets you set *pre-approval* (run without confirmation) and *post-approval* (add the result to context without review) per tool or per MCP server. **Chat: Reset Tool Confirmations** clears saved approvals.
- `chat.tools.eligibleForAutoApproval` with a tool set to `false` forces manual approval for that tool every time.
- `chat.tools.urls.autoApprove` separates request approval from response approval for URLs.
- `chat.tools.terminal.autoApprove` evaluates each terminal sub-command; `true` auto-approves, `false` requires approval (it does not block), and a `false` rule always wins.
- `chat.tools.global.autoApprove` removes prompts in every workspace; prefer the session-scoped **Allow all**.

Listing 2.4: VS Code user settings that loosen safe commands and keep risky ones gated

```
{
  "chat.tools.terminal.autoApprove": {
    "mkdir": true,
    "/^git (status|show\\b.*)$/": true,
    "del": false,
    "/dangerous/": false
  },
  "chat.tools.urls.autoApprove": {
    "https://*.contoso.com/*": true,
    "https://example.com/api/*": { "approveRequest": true, "approveResponse": false }
  }
}
```

> **NOTE**
> Terminal auto-approval is "a best-effort convenience, not a security boundary". Parsers can miss sub-commands and obfuscated commands can evade matching. For high-risk work use sandboxing or a container, and use a `PreToolUse` hook (Preview) when you need to *deny* a call outright in the Local harness.

Copilot CLI: allow, deny and trusted folders

When the CLI needs a modifying tool, it offers three choices: allow once; allow that tool for the rest of the running session; or refuse and tell Copilot what to do differently. The second choice is broad: approving `rm ./this-file.txt` for the session lets Copilot run any `rm` command for the rest of that session.

Command-line rules use `Kind(argument)` patterns:

Kind	Controls	Examples
`shell`	Shell commands	`shell`, `shell(git push)`, `shell(git:*)`
`write`	Creating or modifying files	`write`, `write(.github/copilot-instructions.md)`
`read`	File or directory reads	`read`, `read(.env)`
`url`	URL access via web fetch or shell	`url(github.com)`, `url(https://*.api.com)`
`memory`	Storing facts to agent memory	`memory`
SERVER-NAME	MCP server tools	`MyMCP(create_issue)`, `MyMCP`

The `:*` suffix matches the command stem followed by a space, so `shell(git:*)` matches `git push` but not `gitea`. "Deny rules always take precedence over allow rules, even when `--allow-all` is set or a matching approval has been saved".

Listing 2.5: Copilot CLI permission recipes

```
# Allow all git commands except git push
copilot --allow-tool='shell(git:*)' --deny-tool='shell(git push)'

# Allow all reads, and writes to one file only
copilot --allow-tool='read, write(.github/copilot-instructions.md)'

# Allow two tools from one MCP server
copilot --allow-tool='MyMCP(create_issue), MyMCP(delete_issue)'

# Two layers: shrink what the model can see, then pre-approve a subset
copilot --available-tools='bash,edit,view,grep,glob' \
        --allow-tool='shell(git:*)' --deny-tool='shell(git push)'

# Hide web tools completely for a benchmarking session
copilot --excluded-tools='web_fetch, web_search'

# Throwaway sandbox only: all tools, paths and URLs (deny rules still apply)
copilot --allow-all --deny-tool='shell(rm)'
```

Permission persistence matters for questions about "why was I not asked":

- Choices such as "don't ask again in this repo" are saved to `~/.copilot/permissions-config.json`, scoped to the Git repository root (or working directory).
- Permanently approved URL domains go into `allowedUrls` in `~/.copilot/settings.json` and apply across sessions.
- `--allow-tool` and `--deny-tool` apply only to the current session and are not written to `permissions-config.json`.
- `/reset-allowed-tools` revokes session grants (including `/allow-all` or `/yolo`), returns to the start-up flags, and clears saved approvals for the current location.

Trusted directories are the CLI's first boundary: when a session starts, Copilot CLI asks you to confirm that you trust the files in and below the launch directory. GitHub advises not launching from your home directory or from folders with executables you cannot vouch for, and notes that path scoping "is heuristic". Use `--add-dir` (or `/add-dir`) to grant access to further directories; `/list-dirs` shows them. For non-interactive runs, `--trust-folder` trusts the working directory's hooks, workspace MCP servers and repository plugin settings for that one session only; it is *not* a tool or path permission grant.

PRODUCTION NOTE

Administrators can neutralise the permissive flags. When managed settings set `permissions.disableBypassPermissionsMode` to `"disable"`, `--allow-all-tools`, `--allow-all-paths`, `--allow-all-urls`, `--allow-all` and `--yolo` are suppressed at start-up. An unrecognised value fails closed to `"disable"`. For Copilot Business and Copilot Enterprise licences, an enterprise administrator may block these options entirely.

EXAM TIP

Know the three different CLI controls that are often blurred together. **Autopilot** (`--autopilot`) controls *continuation*. **--allow-all / --yolo** controls *approvals*. **--no-ask-user** suppresses *clarifying questions*. None of them implies the others.

2.2 Configure MCP servers

The Model Context Protocol is an open standard that lets an AI client discover and call tools exposed by an **MCP server**. A server can run locally as a process that talks over standard input and output (**stdio**, called **local** on the cloud agent) or remotely over HTTP (**Streamable HTTP**, with **SSE** as an older transport). The Microsoft Learn module for this domain frames governance as three cooperating parts: the **server** exposes tools, the **registry** makes approved servers discoverable, and the **allow list** decides which servers are permitted.

2.2.1 Add an MCP server as a tool to an agent

There are four configuration homes you must recognise on sight. They differ in top-level key, supported transports and secret handling.

Where	File or location	Top-level key	Secrets	Used by
VS Code workspace or profile	`.vscode/mcp.json`, or `mcp.json` in the user profile	`servers`	`inputs` (`${input:id}`), `envFile`	VS Code; forwarded to Copilot harness sessions except servers that need interactive input
Portable workspace file	`.mcp.json` at the workspace root	`mcpServers`	Environment variables (no VS Code inputs)	Copilot harness sessions, Copilot CLI (trusted folders only)
Copilot user file	`$COPILOT_HOME/mcp-config.json`, fallback `~/.copilot/mcp-config.json`	`mcpServers`	Environment variables	Copilot CLI, VS Code Copilot harness
Copilot cloud agent repository	Repository **Settings** > **Copilot** > **MCP servers** (JSON pasted in the UI)	`mcpServers`	Agents secrets and variables prefixed `COPILOT_MCP_`	Copilot cloud agent and Copilot code review
Custom agent profile	`mcp-servers:` in YAML front matter	(YAML of the same schema)	`COPILOT_MCP_` Agents secrets, also `${{ secrets.X }}` / `${{ vars.X }}`	That agent on GitHub.com and Copilot CLI; ignored by IDE custom agents

VS Code `mcp.json`

The VS Code format has three sections: `servers` (required), `inputs` (optional placeholders for secrets) and `sandbox` (optional deny rules on macOS and Linux). A stdio server needs `type` (`"stdio"`) and `command`; `args`, `cwd`, `env`, `envFile`, `dev`, `sandboxEnabled` and `sandbox` are optional. An HTTP server needs `type` (`"http"` or `"sse"`) and `url`, with optional `headers` and `oauth`. For `http`, VS Code "first tries the HTTP Stream transport and falls back to SSE if HTTP is not supported".

Listing 2.6: `.vscode/mcp.json` with a local stdio server, a secret input and a remote OAuth server

```
{
  "inputs": [
    {
      "type": "promptString",
      "id": "perplexity-key",
      "description": "Perplexity API Key",
      "password": true
    }
  ],
  "servers": {
    "perplexity": {
      "type": "stdio",
      "command": "npx",
```

```
      "args": ["-y", "server-perplexity-ask"],
      "env": { "PERPLEXITY_API_KEY": "${input:perplexity-key}" }
    },
    "slack": {
      "type": "http",
      "url": "https://mcp.slack.com/mcp",
      "oauth": { "clientId": "example-client-id" }
    }
  }
}
```

When VS Code starts the `perplexity` server for the first time it prompts for the key and stores it securely, so nothing sensitive is committed. With `oauth`, VS Code opens a browser for authorisation on first connection. Recognise the launchers in `command`: `npx` runs an npm package (`-y` only auto-confirms the install prompt), `uvx` runs a Python package, `docker` runs a container (never with `-d`, because a stdio server must stay in the foreground). Neither `npx` nor `uvx` pins a version unless you add one to the package specifier.

Trust. Workspace MCP servers inherit Workspace Trust: in a trusted workspace, servers in `.vscode/mcp.json` and `.mcp.json` start without a separate prompt; in Restricted Mode they do not start. Servers from other sources show a trust dialog on first start or after a configuration change; **MCP: Reset Trust** resets those decisions. You can add servers through **MCP: Add Server**, the Agent Customizations editor, the `@mcp` gallery in the Extensions view (**Install in Workspace** writes `.vscode/mcp.json`), `devcontainer.json` (`customizations.vscode.mcp`) or the command line (`code --add-mcp "{...}"`). Once running, the server's tools appear in the tools picker and can be listed in a custom agent as `<server name>/*`.

> **NOTE**
> The Microsoft Learn module walks through the UI path: open Copilot Chat in Agent mode, select the tools icon, **Configure tools**, **Add MCP server**, choose **HTTP**, enter `https://api.githubcopilot.com/mcp/`, choose workspace or global scope, then authenticate. The result is the same JSON entry you could have typed by hand.

Copilot cloud agent repository MCP configuration

Repository administrators paste JSON into **Settings** > **Copilot** > **MCP servers** (under "Code, planning, and automation") and click **Save MCP configuration**; GitHub validates the syntax. This configuration is shared by Copilot cloud agent and Copilot code review. Keys:

- **Required for every server**: `tools` (array of tool names, or `"*"`) and `type` (`"local"`, `"stdio"`, `"http"` or `"sse"`).
- **Local servers**: `command` and `args` (both required), `env` (optional).
- **Remote servers**: `url` (required), `headers` (optional).
- Values in `env` and `headers` are either literals or substitution references such as `$COPILOT_MCP_API_KEY`, `${COPILOT_MCP_API_KEY}` or `${COPILOT_MCP_API_KEY:-default}`. Referenced names must start with `COPILOT_MCP_`. Every string field except `tools` and `type` supports substitution.

Listing 2.7: Repository MCP configuration for Copilot cloud agent (local and remote servers, secrets by reference)

```
{
  "mcpServers": {
    "sentry": {
      "type": "local",
      "command": "npx",
      "args": ["@sentry/mcp-server@latest", "--host=$SENTRY_HOST"],
      "env": {
        "SENTRY_HOST": "https://contoso.sentry.io",
        "SENTRY_ACCESS_TOKEN": "$COPILOT_MCP_SENTRY_ACCESS_TOKEN"
      },
      "tools": ["get_issue_details", "search_issues"]
    },
    "inventory": {
```

```
      "type": "http",
      "url": "https://mcp.contoso.example/mcp",
      "headers": {
        "Authorization": "Bearer ${COPILOT_MCP_INVENTORY_TOKEN}"
      },
      "tools": ["*"]
    }
  }
}
```

The Sentry tool names in Listing 2.7 and the `inventory` server are illustrative; the substitution syntax and keys are as documented. The secrets themselves live in **Settings** > **Secrets and variables** > **Agents** (repository) or the same path at organisation level, which supports **All repositories**, **Private repositories** or **Selected repositories** access. Rules worth memorising:

- Only Agents secrets and variables whose names start with `COPILOT_MCP_` are available to the MCP configuration, and those prefixed values are *only* available to MCP servers, not to the agent's shell.
- Other Agents secrets and variables are exposed to the agent as environment variables (including inside `copilot-setup-steps.yml`), and secret values are masked in session logs.
- The cloud agent does **not** receive Actions, Codespaces or Dependabot secrets. Secrets that used to live in the `copilot` Actions environment were migrated automatically to the repository-level Agents type; older articles may still say "Copilot environment".
- Names may contain only `A-Z`, `0-9` and underscores, must not start with `GITHUB_` or a digit, and a repository-level value overrides an organisation-level value with the same name.

COMMON TRAP

The `${{ secrets.NAME }}` and `${{ vars.NAME }}` syntaxes are supported in the **custom agent YAML** `mcp-servers` block, not in the repository JSON. In the repository JSON use `$COPILOT_MCP_NAME`, `${COPILOT_MCP_NAME}` or `${COPILOT_MCP_NAME:-default}`. And never paste a literal token: literals are allowed syntactically but are stored in plain text.

Important cloud agent limitations: it supports MCP **tools only** (not resources or prompts), and it does **not** support remote MCP servers that use **OAuth** for authentication. Use a token header or a local server instead. For Copilot code review, every tool must advertise `annotations.readOnlyHint: true` or code review excludes it; a repository setting lets you disable MCP tools for code review altogether.

To validate, assign an issue to Copilot, open the pull request, select **View session**, then in the log viewer's **...** menu choose **Copilot** and expand **Start MCP Servers**; started servers list their tools at the bottom. Servers that need tools not on the runner (for example `uv` or `pipx`) need a `copilot-setup-steps.yml` to install them (2.3.6).

Reusing a VS Code configuration on GitHub

GitHub documents four edits when you port `.vscode/mcp.json` to repository settings: add a `tools` key to each server; replace `inputs` with `env`; replace `envFile` with `env`; and update `args` that referenced inputs to use environment variables instead. You also rename the top-level `servers` to `mcpServers`. The same rename applies when migrating `.vscode/mcp.json` to `.mcp.json` for Copilot CLI.

`mcp-servers` in a custom agent profile

Organisation and enterprise administrators (and repository authors) can attach MCP servers to a single agent. The YAML mirrors the JSON schema; `stdio` is mapped to the cloud agent's `local` type for compatibility.

Listing 2.8: Agent profile that brings its own MCP server and enables one of its tools

```
---
name: incident-triage
description: Correlates production errors with recent commits and drafts a fix plan
tools: ["read", "search", "custom-mcp/tool-1"]
```

```
mcp-servers:
  custom-mcp:
    type: 'local'
    command: 'some-command'
    args: ['--arg1', '--arg2']
    tools: ["*"]
    env:
      ENV_VAR_NAME: ${{ secrets.COPILOT_MCP_ENV_VAR_VALUE }}
---

Look up the incident, find the commits that touched the failing code path,
and write a remediation plan. Do not modify source files.
```

Note the double filter in Listing 2.8: the server-level `tools: ["*"]` exposes every tool the server offers, but the agent-level `tools` list enables only `custom-mcp/tool-1`. MCP servers in a profile are "available only to that specific agent".

Copilot CLI MCP management

The CLI persists servers in `~/.copilot/mcp-config.json` and also loads workspace `.mcp.json` and `.github/mcp.json` from the working directory up to the Git root, provided the folder is trusted. In prompt mode (`-p`) workspace MCP sources are not loaded unless you set `GITHUB_COPILOT_PROMPT_MODE_WORKSPACE_MCP=true` or pass `--trust-folder`, which prevents an untrusted repository from starting its own servers in CI. Priority, highest first: `--additional-mcp-config`, plugin servers, workspace servers, user config. In the CLI "all MCP tool invocations require explicit permission", which is why `--allow-tool='SERVER(tool)'` exists.

Listing 2.9: Adding and inspecting MCP servers in Copilot CLI

```
# Local (stdio) server: command after --
copilot mcp add --env SENTRY_HOST=https://contoso.sentry.io sentry -- npx @sentry/mcp-server@latest

# Remote HTTP server with a header and a tool filter
copilot mcp add --transport http --header "X-Team: payments" --tools "*" inventory
https://mcp.contoso.example/mcp

# Inspect, disable, remove
copilot mcp list
copilot mcp get inventory
copilot mcp disable inventory
copilot mcp remove inventory

# One-off server for a single session only
copilot --additional-mcp-config @./ci-mcp.json -p "Summarise open incidents" --allow-tool='inventory'
```

2.2.2 Configure the GitHub remote MCP server

GitHub hosts the GitHub MCP server at `https://api.githubcopilot.com/mcp/`, so there is "no local setup or runtime required". The remote server also exposes tools the local server does not, such as `create_pull_request_with_copilot` for delegating to the cloud agent, and remote-only toolsets such as `copilot_spaces` and `github_support_docs_search`.

Toolsets, read-only mode and headers

Tools are grouped into **toolsets** (for example `actions`, `code_security`, `copilot`, `dependabot`, `discussions`, `gists`, `git`, `issues`, `labels`, `notifications`, `orgs`, `projects`, `pull_requests`, `repos`, `secret_protection`, `security_advisories`, `users`). When no toolsets are specified, the **default** configuration is used: `context`, `repos`, `issues`, `pull_requests` and `users`. You can choose toolsets by URL path or by header:

Mechanism	Example	Notes
Default toolsets	`/mcp/`	context, repos, issues, pull_requests, users

Mechanism	Example	Notes
Read-only	`/mcp/readonly`, `/mcp/x/issues/readonly`	Append `/readonly` to any URL
One toolset	`/mcp/x/copilot`	`{toolset}` must be a single toolset, not a comma list
All toolsets	`/mcp/x/all`	Combine with `/readonly` as needed
Insiders	`/mcp/insiders` or header	Early-access tools
`X-MCP-Toolsets`	`repos,issues`	Several toolsets; unknown names are silently ignored
`X-MCP-Tools`	`get_file_contents,issue_read`	Individual tools; an invalid tool name stops the server from starting
`X-MCP-Readonly`	`true`	Only read tools
`X-MCP-Lockdown`	`true`	Hides public issue details from users without push access; "a best-effort content filter, not a security boundary"

Authentication: OAuth or PAT

In VS Code 1.101 or later, the simplest configuration is just the URL; VS Code runs the OAuth flow. Each MCP host must register its own GitHub App or OAuth App to support OAuth, but "any host application that supports remote MCP servers should support the remote GitHub server with PAT authentication". With a PAT, send it as a bearer token in the `Authorization` header.

Listing 2.10: VS Code configuration for the remote GitHub MCP server with a PAT, two toolsets, read-only

```
{
  "inputs": [
    {
      "type": "promptString",
      "id": "github_mcp_pat",
      "description": "GitHub Personal Access Token",
      "password": true
    }
  ],
  "servers": {
    "github": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp/",
      "headers": {
        "Authorization": "Bearer ${input:github_mcp_pat}",
        "X-MCP-Toolsets": "issues,pull_requests",
        "X-MCP-Readonly": "true"
      }
    }
  }
}
```

If the PAT is already in an environment variable, VS Code's `${env:GITHUB_PAT}` variable can replace the input reference. Prefer a **fine-grained** PAT limited to the repositories and permissions the agent needs; the token, not the prompt, is the real access boundary.

Customising the built-in GitHub MCP server on the cloud agent

On the cloud agent, the built-in `github` server is read-only and scoped to the current repository. To reach other repositories or more toolsets, add a server entry in the repository MCP configuration and supply a broader credential:

Listing 2.11: Cloud agent repository configuration widening the GitHub MCP server

```
{
  "mcpServers": {
    "github-mcp-server": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp/readonly",
      "tools": ["*"],
      "headers": {
        "X-MCP-Toolsets": "repos,issues,users,pull_requests,code_security,actions"
      }
    }
  }
}
```

For PAT authentication, the documented step is to create an Agents secret named exactly `COPILOT_MCP_GITHUB_PERSONAL_ACCESS_TOKEN`, ideally a fine-grained PAT with read-only permissions on specific repositories. Removing `/readonly` enables wider (write) access; use the `X-MCP-Toolsets` header to choose toolsets and the `tools` field to pick individual tools. For a GitHub App, run the local server image `ghcr.io/github/github-mcp-server` with `GITHUB_APP_ID`, `GITHUB_APP_INSTALLATION_ID` and `GITHUB_APP_PRIVATE_KEY` mapped from `COPILOT_MCP_` variables and secrets, plus `GITHUB_READ_ONLY` and `GITHUB_TOOLSETS`. If the session shows a firewall warning for `api.github.com`, add it to the custom allowlist.

COMMON TRAP

If an agent read a repository it should not touch, look at the credential first. A PAT with organisation-wide access lets the GitHub MCP server read any repository the PAT can read, whatever the agent instructions say. The enforceable fix is a fine-grained token limited to the intended repository, stored as an Agents secret. Changing `GITHUB_TOKEN` permissions in a workflow does not affect a separately supplied MCP token.

Hosting a server remotely does not make every client identical. Copilot Chat on GitHub.com comes "preconfigured with tools" for tasks such as creating branches and issues; VS Code, Copilot CLI and the cloud agent each read their own MCP configuration. And for any of them to use MCP under an organisation-managed seat, the MCP policy must allow it (2.2.4).

2.2.3 Configure MCP registries PREVIEW

An **MCP registry** is a catalogue of MCP servers that compatible clients query to show developers which servers they can install. IDEs use the GitHub MCP Registry by default; organisations and enterprises can point Copilot at a custom registry. GitHub labels custom registries and registry-based restriction as public preview and "not the recommended method for restricting access to MCP servers"; the generally available method is the `managed-settings.json` allowlist in 2.2.4. Expect the exam to test both.

Building a registry

Options: fork and self-host the open-source MCP Registry, run it locally with Docker, publish your own implementation, or use **Azure API Center** as a fully managed registry (allow anonymous access in its visibility settings). A valid registry follows the **v0.1** MCP registry specification (v0 is unstable and should not be implemented) and serves:

- `GET /v0.1/servers`
- `GET /v0.1/servers/{serverName}/versions/latest`
- `GET /v0.1/servers/{serverName}/versions/{version}`

All `/v0.1/servers` endpoints must return CORS headers: `Access-Control-Allow-Origin: *`, `Access-Control-Allow-Methods: GET, OPTIONS` and `Access-Control-Allow-Headers: Authorization, Content-Type`. To allow local servers under a restrictive policy, include them with a server ID that exactly matches the installed server ID.

Publishing the registry URL

- **Enterprise**: **AI controls** > **MCP**; set **MCP servers in Copilot** to **Enabled everywhere**; enter the **MCP Registry URL** and save.
- **Organisation**: **Settings** > **Copilot** > **Policies**; in "Features" ensure **MCP servers in Copilot** is **Enabled**; fill **MCP Registry URL (optional)**.
- For Azure API Center, enter the base URL including the workspace path (`https://SERVICE-NAME.data.REGION.azure-apicenter.ms/workspaces/WORKSPACE-NAME`). Do not append `/v0.1/servers`; Copilot adds it and the extra suffix makes the registry error.

Registry display and enforcement are supported in Copilot CLI, VS Code, Visual Studio, JetBrains, Eclipse and Xcode (minimum versions apply). Individual developers can also change the registry their IDE browses, which is useful for testing before a policy is set.

2.2.4 Configure MCP allow lists

Allowlisting happens at three levels, from coarse to fine.

Level	Control	What it decides
Policy switch	**MCP servers in Copilot** policy (enterprise or organisation); for the cloud agent the organisation also has the **MCP servers on GitHub.com** policy	Whether MCP servers can run at all. Third-party MCP use is disabled by default for Copilot Business and Copilot Enterprise members until enabled
Server allowlist (GA, recommended)	`allowedMcpServers` / `deniedMcpServers` in the enterprise `managed-settings.json`	Which servers may run, matched securely by name, URL or exact `stdio` command
Registry restriction (Preview)	**Restrict MCP access to registry servers**: **Allow all** or **Registry only**	Only servers listed in the registry may run; matching by name or ID only
Tool allowlist	`tools` array per server, and agent profile `tools`	Which tools of an allowed server are callable

Listing 2.12: Enterprise `managed-settings.json` allowlist and denylist

```
{
  "allowedMcpServers": [
    { "serverUrl": "https://api.githubcopilot.com/*" },
    { "serverCommand": ["npx", "@playwright/mcp@latest"] },
    { "serverCommand": ["cmd", "/c", "uvx", "markitdown-mcp"] }
  ],
  "deniedMcpServers": [
    {
      "serverCommand": ["npx", "-y", "@modelcontextprotocol/server-filesystem", "/"]
    }
  ]
}
```

Most enterprises store `managed-settings.json` in a `.github-private` repository; it can also be deployed with mobile device management. Clients evaluate each server in this order:

1. Always allow built-in default servers such as the built-in GitHub MCP server.
2. Block the server if it matches any `deniedMcpServers` entry.
3. If `allowedMcpServers` exists, block the server unless it matches an entry.
4. Block a server whose URL or command contains an unresolved variable such as `${VARIABLE}`, because it cannot be verified.

Deny from any layer wins; when several layers define allowlists, a server must match every one. A malformed list is treated as an empty `allowedMcpServers`, blocking everything except built-in servers. If a layer cannot be retrieved, the client keeps the previously enforced policy, so the effective policy can tighten but never loosen. Copilot CLI's enterprise check is

likewise **fail-closed**: it fingerprints each non-default server (command, arguments, remote URL), asks the enterprise endpoint, and blocks unapproved servers with "MCP server "SERVER-NAME" was blocked by your enterprise".

> **EXAM TIP**
> "Registry only" plus a registry is the answer when the question emphasises *discovery* of an approved catalogue. `managed-settings.json` is the answer when the question emphasises *secure enforcement that users cannot bypass by editing local configuration*. Registry enforcement "is based only on server name/ID matching, which can be bypassed by editing configuration files".

When a user holds several Copilot seats, registry policy resolves deterministically: enterprise policy overrides organisation policy; **Registry only** beats **Allow all**; with equal scope and strictness the most recently uploaded registry applies. If you migrate from a registry to `managed-settings.json`, GitHub recommends setting the registry restriction back to **Allow all** to keep a single source of truth.

2.3 Integrate agents within development environments

The same model and the same custom agent can run in very different places. The **execution context** decides what the agent can see, which credentials it holds, whether a human is watching, and how its output reaches the codebase. The Learn module defines execution context as the repository the agent works in, the branch it targets, the workflow that executes the task, and the permissions granted to that workflow.

2.3.1 Evaluate execution context

Context	Where it runs	Human in the loop	Best for	Watch out for
VS Code, Local harness	VS Code extension host on your machine, tied to the editor window	High: approvals, pending edits with **Keep** / **Undo**	Interactive implementation, debugging, work that needs local resources or extensions	Closing the window stops the work
VS Code, Copilot harness	Copilot SDK in the Agent Host; continues in the background after the window closes while the host runs; folder or isolated Git worktree	Medium: permission levels, Autopilot available	Longer local tasks, parallel sessions, sessions continued across interfaces	Not every VS Code built-in or extension tool is available
Copilot CLI (interactive)	Your terminal, in a trusted directory	High: per-tool prompts, `/diff`, `/undo`	Terminal-centric work, scripting, remote shells	Session-wide approvals are broad
Copilot CLI (programmatic, `-p`)	Any shell, including a GitHub Actions runner	None during the run; anything not pre-approved is denied	Scheduled reports, CI checks, scripted fixes	Credentials, permissions and fork-triggered workflows
Copilot cloud agent (Cloud target)	Ephemeral GitHub Actions-powered environment	At pull request review; steering via comments or the agents page	Well-scoped issues, background work, team-visible changes	Firewall, setup steps, secrets, runner choice
GitHub Agentic Workflows	GitHub Actions, compiled from Markdown with front matter	Via safe outputs and gated write steps	Continuous repository automation (triage, reports, documentation)	Separate tooling (`gh-aw`); read-only by default

Use three questions to choose: **Where are the resources?** (a local database or device points to a local context; a GitHub repository and CI points to the cloud agent); **who needs to see the result?** (a pull request reviewed by the team points to the cloud agent; a private experiment points to a local session); and **how much autonomy is acceptable?** (unattended work belongs in an isolated, ephemeral environment with least-privilege credentials). In VS Code, choose the target with the **Session Target** control: Copilot, Local, Claude, Codex or Cloud. Cloud "is an execution target, not a single harness": a provider works against a GitHub repository and returns a pull request, without local editor context.

> **NOTE**
> VS Code terminology has evolved quickly. Older material talks about "local agents", "background agents (Copilot CLI)" and "cloud agents". The concepts map directly onto the Local harness, the Copilot harness or Copilot CLI running in the background, and the Cloud target. Exam scenarios usually describe the behaviour (keeps running after you close the editor, opens a pull request, needs your local database) rather than the label.

2.3.2 Scope an agent to a specific repository

Repository scope is the first boundary. Configure it at several layers, because each one closes a different gap:

- **Enable the cloud agent only where it belongs.** Organisation owners enable the **Copilot cloud agent** policy, then use **Settings** > **Copilot** > **Cloud agent** > **Repository access** to allow all or **Selected repositories** (also available through the REST API). Afterwards "any user with access to Copilot cloud agent and write permission for the repository can delegate work to Copilot". For personal repositories the setting is in your account's Copilot policies.
- **Rely on the built-in token scope.** The cloud agent pushes only to its own branch and its default GitHub MCP token is read-only and scoped to the source repository. "It can't access other repositories" unless you explicitly widen access.
- **Scope every additional credential.** MCP servers receive whatever you put in `COPILOT_MCP_` secrets. Use fine-grained PATs limited to one repository, or a GitHub App installed on selected repositories. Organisation-level Agents secrets can be restricted to **Selected repositories**.
- **Choose where agent profiles live.** A profile in `.github/agents/` applies to that repository; profiles in an organisation's `.github` or `.github-private` repository apply across the organisation. A repository profile with the same file name overrides the organisation one.
- **Scope local tools to a directory.** Copilot CLI trusts only the launch directory and those added with `--add-dir`; VS Code workspace MCP servers require Workspace Trust.

> **COMMON TRAP**
> The Learn module illustrates file focus with an `applyTo` list inside an agent file. In GitHub Docs, `applyTo` is the front matter of path-specific `*.instructions.md` files; the custom agents configuration reference does not list it for agent profiles. Within a repository, the documented, enforceable levers for an agent are its `tools` list, its MCP configuration and the credentials it receives. Instructions in the body guide focus but do not enforce it.

2.3.3 Invoke an agent in a CI workflow

Copilot CLI runs on an Actions runner like any CLI. The documented pattern is: **trigger** (schedule, event or `workflow_dispatch`), **setup** (checkout, runtime), **install** (`npm install -g @github/copilot`), **authenticate**, and **run** `copilot -p PROMPT [OPTIONS]`. GitHub recommends **GitHub Agentic Workflows** for most automation because they use `GITHUB_TOKEN` by default and add guardrails, but you must also know the direct pattern.

Authentication choices

Option	How	When
`GITHUB_TOKEN` (recommended for organisation-owned repositories)	Grant `copilot-requests: write` in `permissions` and pass `GITHUB_TOKEN: ${{ github.token }}`; the organisation policy **Allow use of Copilot CLI billed to the organization** must be selected (on by default where Copilot CLI is enabled)	No stored secret; usage billed to the organisation
Fine-grained PAT	PAT with the **Copilot Requests** permission, stored as an Actions secret, passed as `COPILOT_GITHUB_TOKEN`	The organisation policy is off, or usage must be billed to a specific user's seat

Token precedence in the CLI is `COPILOT_GITHUB_TOKEN`, then `GH_TOKEN`, then `GITHUB_TOKEN`. The values of `GITHUB_TOKEN` and `COPILOT_GITHUB_TOKEN` are redacted from output by default; add others with `--secret-env-vars`.

Least-privilege permissions

The `permissions` key sets the scopes of the job's `GITHUB_TOKEN`. "If you specify the access for any of these permissions, all of those that are not specified are set to `none`." So declare exactly what the job needs: `contents: read` to read code, `copilot-requests: write` (only `write` and `none` are valid) to call Copilot, and, for example, `issues: write` only if the job files an issue. Repository and organisation settings set the default token permissions; read-only is the safe default.

Listing 2.13: Least-privilege workflow that runs Copilot CLI non-interactively with `GITHUB_TOKEN`

```
name: Copilot change summary
on:
  push:
    branches: [main]
  workflow_dispatch:

permissions:
  contents: read
  copilot-requests: write

jobs:
  summarise:
    runs-on: ubuntu-latest
    timeout-minutes: 15
    steps:
      - uses: actions/checkout@v6
        with:
          fetch-depth: 2

      - name: Install Copilot CLI
        run: npm install -g @github/copilot

      - name: Summarise the latest commit
        env:
          GITHUB_TOKEN: ${{ github.token }}
          COPILOT_AUTO_UPDATE: "false"
        run: |
          copilot -p "Use git show HEAD to summarise the changes in this commit" \
            -s --no-ask-user \
            --allow-tool='shell(git show:*)' \
            --share=./copilot-session.md
          cat ./copilot-session.md >> "$GITHUB_STEP_SUMMARY"
```

Read Listing 2.13 against the documented behaviour:

- `-p` runs one prompt and exits; `-s` prints only the agent response.
- `--no-ask-user` disables the `ask_user` tool because nobody can answer.

- "Non-interactive (`-p`) runs can't display an interactive approval prompt, so any action that isn't pre-approved is denied automatically." Hence the narrow `--allow-tool='shell(git show:*)'`.
- `--share=PATH` exports the transcript for audit; `COPILOT_AUTO_UPDATE=false` pins the installed CLI version for reproducibility.
- To run a custom agent, add `--agent NAME`; to pin behaviour across runs, add `--model`.

COMMON TRAP

GitHub warns that "invoking Copilot CLI directly in workflow steps gives it broad access to your workflow environment" and that "workflows triggered by pull requests from forks are particularly at risk". Do not combine untrusted triggers with write permissions, secrets and `--allow-all`. Treat issue bodies, comments and pull request text as untrusted input that may contain prompt injection.

Recursion is another CI hazard. Events caused by `GITHUB_TOKEN` do not start new workflow runs, except `workflow_dispatch` and `repository_dispatch`, and pull requests opened, synchronised or reopened with `GITHUB_TOKEN`, whose runs start in an approval-required state. A PAT or GitHub App token can trigger further runs, which is useful but can create loops.

2.3.4 Configure branch-based scope

Agents do not work directly on the default branch. The cloud agent's branch rules are documented as risk mitigations:

- "Copilot cloud agent only has the ability to push to a single branch." In most cases "a new `copilot/` branch is created for Copilot, and the agent can only push to that branch".
- When triggered by mentioning `@copilot` on an existing pull request, Copilot has write access to *that pull request's branch* and pushes there by default, unless you ask for a separate pull request.
- "It cannot directly run `git push` or other Git commands"; its credentials allow only simple push operations.
- The agent "is also subject to any branch protections and required checks for the working repository".

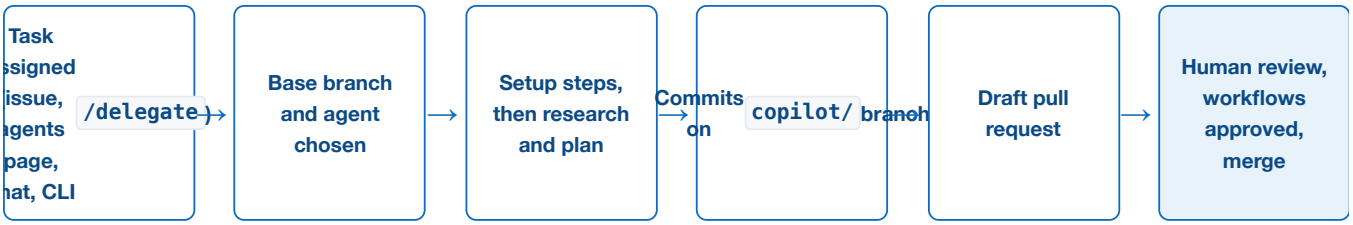
Base branch selection. From the agents page, the agents panel or the **Agents** tab you can optionally select a base branch; Copilot creates its new branch from it, and the pull request targets it. When assigning an issue, the dialog lets you change the repository and the branch to branch off from; if you specify none, Copilot works from the default branch. Custom agents do not control branch behaviour; "branch-based scope is always enforced by the execution context". In a CI workflow, the checked-out ref decides the branch the CLI works on.

EXAM TIP

Pair branch scope with rulesets. Required reviews, required status checks and restrictions on the default branch keep working when the author is an agent. That is what makes `copilot/` branches a safe sandbox rather than a convention.

2.3.5 Enable autonomous actions, including creating branches and pull requests

Autonomy is enabled by the platform, then bounded by settings. The end-to-end flow:



Prerequisites: the **Copilot cloud agent** policy is enabled for the organisation's members (it is disabled by default for Business and Enterprise seats), the repository is within the allowed repository access, and the person delegating has write access. Ways to start work include assigning an issue to Copilot, the agents page at `github.com/copilot/agents`, the

Agents tab of a repository, Copilot Chat, mentioning `@copilot` in a pull request, asking Copilot to fix a failing Actions run, Copilot CLI's `/delegate`, the remote GitHub MCP server's `create_pull_request_with_copilot` tool, and **automations** that run on a schedule or event. When you assign an issue, Copilot reacts with an eyes emoji and soon opens a draft pull request linked to the issue; from the agents page you can review changes before creating a pull request or ask for one in the initial prompt.

Built-in limits on that autonomy:

- Only users with write access can trigger the agent; comments from others are never presented to it. Hidden characters, such as HTML comments in issues, are filtered out to reduce prompt injection.
- Copilot "cannot mark its pull requests as Ready for review and cannot approve or merge a pull request".
- The user who asked Copilot to create the pull request cannot approve it, preserving "Required approvals". When Copilot opens a pull request under its own app identity, one more approval is required (on by default in rulesets, always for branch protection rules).
- By default GitHub Actions workflows do not run on Copilot's pushes until someone with write access clicks **Approve and run workflows**. A repository administrator can disable **Require approval for workflow runs** under **Settings** > **Copilot** > **Cloud agent**, with the warning that unreviewed code could gain write access or reach Actions secrets.
- Automations attribute work to the user who created them, ignore events from users without write access by default, and let you choose exactly which tools they can use.

Built-in validation also runs by default: CodeQL for code security, the GitHub Advisory Database for new dependencies, secret scanning, and a Copilot code review "second opinion". Repository administrators can toggle these under **Validation tools**; no GitHub Advanced Security licence is required.

2.3.6 Apply environment-specific constraints

`copilot-setup-steps.yml`

The cloud agent's environment is ephemeral and powered by GitHub Actions. Customise it with `.github/workflows/copilot-setup-steps.yml`, a normal workflow file that must contain a single job named `copilot-setup-steps`. Its steps run before Copilot starts work. Rules to memorise:

- It "won't trigger unless it's present on your default branch".
- Only these job settings are honoured; others are ignored: `steps`, `permissions`, `runs-on`, `services`, `snapshot` and `timeout-minutes` (maximum `59`).
- Set `permissions` as low as possible: "Copilot will be given its own token for its operations". You need `contents: read` only if your steps check out the code; otherwise Copilot clones the repository after the steps.
- Any `fetch-depth` you give `actions/checkout` is overridden "to allow the agent to rollback commits upon request".
- If a step exits non-zero, Copilot skips the remaining setup steps and starts work in the current state of the environment.
- Trigger it on `push` and `pull_request` path filters plus `workflow_dispatch` to validate changes like any other workflow.
- Copilot code review reuses this file by default, or uses a dedicated `copilot-code-review.yml`.

Listing 2.14: `.github/workflows/copilot-setup-steps.yml` preinstalling dependencies on a larger runner

```yaml
name: "Copilot Setup Steps"

on:
  workflow_dispatch:
  push:
    paths:
      - .github/workflows/copilot-setup-steps.yml
  pull_request:
    paths:
      - .github/workflows/copilot-setup-steps.yml
```

```yaml
jobs:
  # The job MUST be called copilot-setup-steps
  copilot-setup-steps:
    runs-on: ubuntu-4-core
    timeout-minutes: 30
    permissions:
      contents: read
    steps:
      - name: Checkout code
        uses: actions/checkout@v6

      - name: Set up Node.js
        uses: actions/setup-node@v7
        with:
          node-version: "20"
          cache: "npm"

      - name: Install JavaScript dependencies
        run: npm ci

      - name: Install uv for Python-based MCP servers
        run: pipx install uv
```

The last step in Listing 2.14 is an illustrative way to provide `uv`, which GitHub cites as an example of an MCP dependency not installed on runners by default. Setup steps that need cloud credentials can use OIDC: the documented Azure example grants `id-token: write` and runs `azure/login` with client and tenant IDs stored as Agents secrets.

Runners

Runner choice	How	Constraints
Standard GitHub-hosted (default)	Nothing to configure; Ubuntu Linux	Integrated firewall available
Larger GitHub-hosted runners	Create them for the organisation, then set `runs-on` to their label or group (for example `ubuntu-4-core`)	With Azure private networking, allow outbound access to `uploads.github.com`, `user-images.githubusercontent.com` and the relevant `api.*.githubcopilot.com` host
Self-hosted runners	Set `runs-on` to an ARC scale set name; prefer ephemeral, single-use runners	You must **disable the integrated firewall** (incompatible) and apply your own network controls; optional proxy variables `https_proxy`, `http_proxy`, `no_proxy`, `ssl_cert_file`, `node_extra_ca_certs`
Windows	Windows label on larger or self-hosted runners	Firewall not compatible; use self-hosted or Azure private networking controls

The cloud agent supports only Ubuntu x64 and Windows 64-bit runners; macOS is not supported. Organisation owners can set a default runner type for all repositories and choose whether repositories may override it. Managed user accounts cannot use the cloud agent in personal repositories, because GitHub-hosted runners are not available there.

Firewall and allowlist

By default the agent's internet access is limited by a firewall to reduce exfiltration risk. GitHub hosts that Copilot needs are always allowed, and a **recommended allowlist** (OS package repositories, container registries, language package registries, certificate authorities, Playwright browser downloads) is enabled by default. Configure it under **Settings** > **Copilot** > **Internet access**, which has separate sections for the cloud agent and code review.

- **Organisation level**: **Enable firewall** and **Recommended allowlist** each take **Enabled**, **Disabled** or **Let repositories decide** (default); **Allow repository custom rules** (default **Enabled**); an **Organization custom allowlist** that repositories cannot delete. Repository settings are locked unless the organisation lets repositories decide.
- **Entries**: a domain allows that domain and its subdomains; a URL allows only that scheme, host and path prefix.

- **Blocked requests** add a warning to the pull request body or a comment, naming the blocked address and the command.
- **Limits**: the firewall applies only to processes the agent starts through its Bash tool, not to MCP server processes or setup steps, and only inside the GitHub Actions appliance; sophisticated attacks may bypass it.

> **PRODUCTION NOTE**
> Because setup steps and MCP servers are outside the firewall, review them with the same care as production workflows. Pin action versions, keep their permissions minimal, and prefer read-only MCP tools.

Secrets and variables in the agent environment

Use **Agents** secrets and variables (repository or organisation level) for anything the agent or its setup steps need: private registry tokens, proxy settings, MCP keys. Names prefixed `COPILOT_MCP_` go only to MCP servers; all others become environment variables for the agent and `copilot-setup-steps.yml`. Values are masked in session logs. The agent cannot see Actions, Codespaces or Dependabot secrets.

2.4 Operate agents with safe execution paths and robust error handling

Agents are non-deterministic. A safe execution path assumes that any run can fail, loop, half-finish or produce a plausible but wrong change, and designs for detection, bounded recovery, reversal and human escalation. The Learn module summarises the goal as execution that is **resilient** (handles failures and retries), **controlled** (prevents unsafe changes), **auditable** (all actions visible and traceable) and **human-governed** (escalation keeps oversight).

2.4.1 Error handling

Make failures loud, early and diagnosable.

Context	Where errors surface	What to do
CI step running Copilot CLI	Step exit status, job log, transcript from `--share`	Fail fast (do not set `continue-on-error: true` for the agent step), validate outputs explicitly, enforce `timeout-minutes` or a `timeout` wrapper
Copilot CLI interactive	Terminal, logs in `~/.copilot/logs/` (`--log-dir`, `--log-level`)	Press `Esc` twice to interrupt a turn; `Ctrl+C` stops Autopilot; review with `/diff`
Cloud agent setup	Session log, setup steps section	A non-zero setup step makes Copilot skip the remaining steps and start anyway, so make critical installs idempotent and check the log
Cloud agent network	Firewall warning in the pull request body or a comment	Add the host to the custom allowlist, or install the dependency in setup steps
MCP servers	Cloud agent: **Start MCP Servers** log step; VS Code: error indicator in chat and **Show Output**; CLI: `copilot mcp list` status and warnings for skipped workspace entries	Fix the configuration; remember CLI skips only invalid entries but ignores a malformed file entirely
Stuck cloud agent session	No progress in the session log	Sessions time out after an hour; steer, stop, or retry

Validate what the agent produced, not just whether the process ended. A `-p` run can exit cleanly having done nothing useful, so check for the expected file, test results or diff before you consume them. In non-interactive runs, actions that were not pre-approved are denied automatically; a run that "did nothing" often means a missing `--allow-tool`, which you can see in the transcript.

Bound runtime and cost explicitly. `--max-autopilot-continues` caps automatic continuation messages (default unlimited); `--max-ai-credits` sets a soft AI credit limit per response; `COPILOT_TASK_WAIT_TIMEOUT_SECONDS` (default `600`) limits how long `-p` waits for background agents or shell commands before exiting. None of these is a wall-clock limit for the whole job, so keep `timeout-minutes` on the job.

2.4.2 Retries

Retry transient failures (network, rate limits, a flaky dependency download) a bounded number of times with back-off. Do not blindly retry logical failures: if the agent misunderstood the task, the same prompt will usually fail the same way, so change the prompt, the tools or the environment first. Make retried work idempotent: write to a fresh branch or file per attempt, or start each attempt from a clean checkout.

Listing 2.15: Bounded retry with validation, transcripts and escalation to an issue

```
name: Nightly dependency audit
on:
  schedule:
    - cron: '0 3 * * 1-5'
  workflow_dispatch:

permissions:
  contents: read
  copilot-requests: write
  issues: write

jobs:
  audit:
    runs-on: ubuntu-latest
    timeout-minutes: 40
    steps:
      - uses: actions/checkout@v6
      - name: Install Copilot CLI
        run: npm install -g @github/copilot

      - name: Run agent with bounded retries
        env:
          GITHUB_TOKEN: ${{ github.token }}
        run: |
          set -uo pipefail
          for attempt in 1 2 3; do
            rm -f audit.md
            if timeout 600 copilot -p "Run npm audit and write high or critical findings to audit.md" \
                 -s --no-ask-user \
                 --allow-tool='shell(npm audit:*), write(audit.md)' \
                 --deny-tool='shell(git push)' \
                 --share="./session-attempt-${attempt}.md" && test -s audit.md; then
              exit 0
            fi
            echo "Attempt ${attempt} failed" >&2
            sleep $((attempt * 30))
          done
          exit 1

      - name: Keep transcripts for audit
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: copilot-sessions
          path: session-attempt-*.md

      - name: Escalate to maintainers
        if: failure()
        env:
          GH_TOKEN: ${{ github.token }}
        run: |
          gh issue create \
            --title "Agent task failed: nightly dependency audit" \
```

```
            --body "Three attempts failed. Run: ${{ github.server_url }}/${{ github.repository
}}/actions/runs/${{ github.run_id }}"
```

Listing 2.15 is a pattern assembled from documented primitives: least-privilege `permissions` (with `issues: write` only because the job escalates through an issue), a narrow pre-approval, an explicit deny, a per-attempt timeout, output validation, back-off, transcripts preserved with `if: always()`, and a human escalation path with `if: failure()`. Because events created with `GITHUB_TOKEN` do not start new workflow runs, opening the issue cannot trigger a retry loop.

Retry options on the other surfaces:

- **Cloud agent**: if a session stays stuck it times out after an hour; "you can retry by unassigning the issue and then reassigning it to Copilot". If it stalled while responding to a comment, add the same comment again. Mid-session, steer it with a follow-up prompt on the agents page (each steering message consumes AI credits); Copilot applies it after its current tool call.
- **Copilot CLI**: a `-p` run prints a `copilot --resume=SESSION-ID` hint; `--continue` resumes the most recent session in the current directory. `/compact` summarises long history if the context window is the problem.
- **VS Code Autopilot** retries automatically when it hits errors; use it only in trusted workspaces.

2.4.3 Rollbacks

Rollback is cheapest before merge. Branch isolation and pull request review mean most bad agent output never reaches the default branch.

Situation	Rollback mechanism
VS Code Local harness, edits not yet kept	**Undo** on pending edits (or **Keep** to accept)
VS Code Copilot harness (edits saved directly)	Checkpoints or Git; worktree isolation keeps your main folder clean
Copilot CLI session	`/undo` or `/rewind`: **Conversation only** or **Conversation + files** (restores files Copilot changed)
Cloud agent work in progress	**Stop session** (ends the Actions run, keeps pushed commits), ask Copilot in the pull request to revert specific commits, or push your own fix
Unwanted pull request	Close it and delete the `copilot/` branch; nothing reached the base branch
Merged pull request	Click **Revert** on the merged pull request, which creates a new pull request reverting the merge commit (requires write permission), then review and merge it
Revert conflicts, or changes not merged through GitHub	`git revert` individual commits on a new branch and open a pull request

Listing 2.16: Reverting merged agent work with Git (through a reviewed pull request)

```
# Start from the current default branch
git switch main && git pull --ff-only
git switch -c revert/agent-change

# Revert the merge commit of the agent's pull request (keep the main line as parent 1)
git revert -m 1 <merge-commit-sha>

# Or revert individual agent commits, newest first
git revert --no-edit <sha-3> <sha-2> <sha-1>

git push -u origin revert/agent-change
gh pr create --fill --base main
```

Prefer reverting over rewriting history: `git revert` adds new commits, so the audit trail of what the agent did, and what you undid, remains intact. Avoid force-pushing protected branches. Each cloud agent commit links to its session log, which helps you pick the right commits to revert.

2.4.4 Escalation paths

Escalation means the system knows when to stop and hand a decision to a human. Build escalation in at four points:

1. **Before acting**: use plan modes for complex work (VS Code Plan, CLI `/plan` or `--plan`) and review the plan. `--plan --mode autopilot` skips the plan approval, so do not use it where a human must sign off.
2. **While acting**: keep approval prompts for risky tools (manual permissions, `chat.tools.eligibleForAutoApproval`, CLI prompts). In interactive sessions leave `ask_user` enabled so the agent can ask; in CI, where `--no-ask-user` is required, encode the stop condition in the prompt (for example "if the fix needs a schema change, write a note and stop") and route failures to an issue or notification.
3. **Before merge**: draft pull requests, required reviews, the rule that the requester cannot approve, the extra approval for unattributed Copilot pull requests, automatic reviewer assignment, and **Approve and run workflows** for Copilot's pushes.
4. **Before deploy**: GitHub Actions environments with required reviewers and environment-scoped secrets, so agent-generated changes cannot deploy themselves.

> **COMMON TRAP**
> A script that runs `copilot --autopilot --yolo -p "..."` has no human checkpoints during the run. `--yolo` grants all permissions (subject to deny rules); `-p` is non-interactive; Autopilot keeps going until the task is complete, a problem occurs or the continuation cap is reached. Human oversight must be designed in elsewhere: in review gates, environments and escalation steps.

2.4.5 Traceability and accountability

Every agent action should answer three questions afterwards: *who asked*, *what did the agent do and why*, and *who approved it*.

Evidence	What it shows	Where
Session logs	Internal reasoning, tools used, setup steps, validation results, MCP server start-up	**View session** on the pull request; the agents page; Copilot Chat on GitHub.com can answer questions using them
Commit authorship	Commits "authored by Copilot, with the developer who assigned the issue or requested the change to the pull request marked as the co-author"	Commit history
Commit signing	Agent commits are signed and show as **Verified**	Commit history
Commit message link	Each agent commit message links to its session log	Commit message
Audit log	Agentic events: filter `actor:Copilot` for the last 180 days; fields `action`, `actor_is_agent`, `agent_session_id`, `user`	Enterprise audit log; streaming of Copilot usage records is Preview
Pull request record	Reviews, approvals, workflow approvals and discussion	Pull request timeline
CLI transcripts and logs	Prompts, responses, tool calls	`--share=PATH`, `--share-gist`, `/share`, `~/.copilot/logs/`
Workflow logs	Exact commands, step results, uploaded artifacts	Actions run

Visibility defaults differ: cloud agent sessions are **shared** by default in the repository's "All sessions" view and can be archived but not deleted; local sessions from Copilot CLI, VS Code and JetBrains are unshared by default but can be shared view-only. Automations attribute their pull requests and pushes to the user who created the automation. Secret values are masked in cloud agent session logs, and the CLI redacts `GITHUB_TOKEN` and `COPILOT_GITHUB_TOKEN` by default; add others with `--secret-env-vars`. Transcripts can still contain sensitive data, so store shared transcripts as restricted artifacts. Note that `--share-gist` is not available to Enterprise Managed Users or on GHE.com.

EXAM TIP

For "who started this agent change?" the answer is the co-author on the agent's commits (or the automation's creator), the `user` field in the audit log event, and the pull request's linked session. For "prove the commit really came from the agent and was not altered", the answer is signed, **Verified** commits.

2.5 Key terms

Tool alias	A portable tool name in an agent profile (`read` , `edit` , `search` , `execute` , `agent` , `web` , `todo`) that maps to each product's real tools.
Tool set	A named VS Code group of tools, referenced with `#name` or listed in an agent's `tools` ; user sets are defined in a `.jsonc` file.
MCP server	A local (stdio) or remote (HTTP, SSE) process that exposes tools, and optionally resources and prompts, to an AI client through the Model Context Protocol.
`server/tool`	Agent profile syntax for an MCP tool; `server/*` selects every tool of that server.
`COPILOT_MCP_` prefix	Required name prefix for Agents secrets and variables that the cloud agent's MCP configuration may reference; such values are available only to MCP servers.
Agents secrets and variables	The dedicated secret type for Copilot cloud agent at repository or organisation level, separate from Actions, Codespaces and Dependabot secrets.
Toolset (GitHub MCP)	A group of GitHub MCP server tools, such as `issues` or `pull_requests` , selected by URL path (`/x/{toolset}`) or the `X-MCP-Toolsets` header.
MCP registry	A catalogue of MCP servers served over HTTPS per the v0.1 specification; custom registries and registry-only restriction are in public preview.
MCP allowlist	Policy that limits which MCP servers may run, preferably via `allowedMcpServers` / `deniedMcpServers` in `managed-settings.json` .
Execution context	The repository, branch, workflow, runner and permissions within which an agent runs.
`copilot-setup-steps.yml`	Workflow with a single `copilot-setup-steps` job that prepares the cloud agent's ephemeral environment before work starts.
Agent firewall	Network restriction for processes the cloud agent starts through its Bash tool, with a recommended and custom allowlist.
Trusted directory	A folder Copilot CLI has been told to trust for reading, modifying and executing files; the start of its path-scoping boundary.
Programmatic mode	Running Copilot CLI with `-p` (or piped input): one prompt, no interactive approvals, exits when done.
Session log	The record of a cloud agent session's reasoning, tools and validation, linked from the pull request and from each commit message.

2.6 Additional practice questions

Practice 2.1 MULTIPLE CHOICE

Fabrikam's repository MCP configuration for Copilot cloud agent defines a server named `tracker` with `"tools": ["*"]`. A custom agent profile in `.github/agents/triage.agent.md` contains `tools: ["read", "search", "github/*"]`. When the agent runs, it never calls any `tracker` tool. You need the agent to use only the `tracker` tool named `list_tickets`, without changing anything else it can do. What should you do?

A. Add an instruction to the agent body: "Use the tracker list_tickets tool when triaging."
B. Add `tracker/list_tickets` to the agent's `tools` list.
C. Replace the agent's `tools` list with `tools: []`.
D. Change the repository MCP configuration to `"tools": ["list_tickets"]`.

Practice 2.2 MULTIPLE CHOICE

You add a remote MCP server to a repository's MCP configuration for Copilot cloud agent. The server requires an API key in an `X-Api-Key` header. Company policy forbids storing credentials in plain text. You need to supply the key. Which configuration should you use?

A. Create an Actions repository secret named `OBS_KEY` and set the header to `${{ secrets.OBS_KEY }}`.
B. Create an Agents secret named `COPILOT_MCP_OBS_KEY` and set the header to `$COPILOT_MCP_OBS_KEY`.
C. Create an Agents secret named `OBS_KEY` and set the header to `$OBS_KEY`.
D. Create a Codespaces secret named `COPILOT_MCP_OBS_KEY` and set the header to `${COPILOT_MCP_OBS_KEY}`.

Practice 2.3 MULTIPLE CHOICE

Developers use the remote GitHub MCP server in VS Code. You need to expose only the `issues` and `pull_requests` toolsets, in read-only mode, using a single server entry. What should you configure?

A. `"url": "https://api.githubcopilot.com/mcp/x/issues,pull_requests/readonly"`
B. `"url": "https://api.githubcopilot.com/mcp/"` with headers `X-MCP-Toolsets: issues,pull_requests` and `X-MCP-Readonly: true`
C. `"url": "https://api.githubcopilot.com/mcp/x/all"` with header `X-MCP-Lockdown: true`
D. `"type": "stdio"` with `"command": "https://api.githubcopilot.com/mcp/readonly"`

Practice 2.4 MULTIPLE CHOICE

A nightly job runs Copilot CLI with `-p`. The agent must be able to run any `git` command except pushing, and must never write to `.env`. All other actions must remain unavailable without approval. Which options meet the requirement?

A. `--allow-all --deny-tool='shell(git push)'`
B. `--allow-tool='shell(git:*)' --deny-tool='shell(git push), write(.env)'`
C. `--excluded-tools='git push' --allow-all-tools`
D. `--allow-tool='shell(git push)' --deny-tool='shell(git:*)'`

Practice 2.5 MULTIPLE CHOICE

Contoso's security team requires that developers can run only approved MCP servers in Copilot clients, matched by server URL or exact command, in a way developers cannot bypass by editing their local configuration files. What should you implement?

A. A custom MCP registry with **Restrict MCP access to registry servers** set to **Registry only**.
B. Set the **MCP servers in Copilot** policy to **Disabled**.
C. `allowedMcpServers` and `deniedMcpServers` in the enterprise `managed-settings.json`.
D. A repository custom instruction listing the approved servers.

Practice 2.6 MULTIPLE ANSWER

You create `.github/workflows/copilot-setup-steps.yml` in a feature branch. The job is named `setup`, sets `timeout-minutes: 90`, and installs a private SDK. Copilot cloud agent ignores the setup. Which TWO changes are required for the setup to be used? Each correct answer presents part of the solution.

A. Rename the job to `copilot-setup-steps`.
B. Merge the file into the default branch.
C. Add `permissions: write-all` to the job.
D. Move the file to `.github/copilot-setup-steps.yml`.

Practice 2.7 MULTIPLE CHOICE

Your organisation wants Copilot cloud agent to reach internal package feeds, so you plan to run it on self-hosted Ubuntu x64 runners managed by Actions Runner Controller. What must you also do?

A. Enable the recommended allowlist at organisation level.
B. Disable Copilot cloud agent's integrated firewall for the repository.
C. Switch the runners to macOS images.
D. Add `container:` to the `copilot-setup-steps` job.

Practice 2.8 MULTIPLE CHOICE

An organisation-owned repository runs Copilot CLI in a workflow authenticated with `GITHUB_TOKEN`. The job only reads code and prints a summary. You need least-privilege permissions. Which block should you use?

A. `permissions: write-all`
B. `permissions: { contents: read }`
C. `permissions: { contents: read, copilot-requests: write }`
D. `permissions: { contents: write, pull-requests: write, copilot-requests: read }`

Practice 2.9 MULTIPLE CHOICE

A pull request created by Copilot cloud agent was approved and merged into `main` yesterday. It introduced a regression. You need to undo it quickly while keeping a reviewable, auditable history. What should you do first?

A. Delete the `copilot/` branch that the pull request came from.
B. Force-push `main` to the commit before the merge.
C. Click **Revert** on the merged pull request and merge the resulting pull request.
D. Archive the agent session.

Practice 2.10 MULTIPLE CHOICE

An enterprise auditor asks for every pull request that agents created in the last quarter, with a link back to the agent session that produced each one and the person who initiated it. Where should you look?

A. The enterprise audit log filtered with `actor:Copilot`, using the `agent_session_id` and `user` fields.
B. The Actions usage report for the organisation.
C. Each developer's local `~/.copilot/logs` directory.
D. The repository's `copilot-setup-steps.yml` run history.

Practice 2.11 MULTIPLE CHOICE

A developer needs an agent to refactor a module, run the tests, and hand the result to the team as a pull request. The work will take a long time and the developer wants to shut down their laptop. The task needs only the repository and public package registries. Which execution context is most appropriate?

A. VS Code Local harness in Agent mode.
B. Copilot CLI in interactive mode on the laptop.
C. Copilot cloud agent, started from the agents page with a selected base branch.
D. A VS Code tool set referenced with `#`.

Practice 2.12 MULTIPLE CHOICE

A workflow runs `copilot -p "Fix the lint errors and save the files" -s --allow-tool='shell(npx eslint:*)' --no-ask-user`. The log shows the agent ran ESLint but no files changed. What is the most likely cause?

A. `-s` suppresses file writes.
B. File writes were not pre-approved, so they were denied automatically in non-interactive mode.
C. `--no-ask-user` disables all tools.
D. The firewall blocked writes to the workspace.

Answers and explanations

2.1: B. A specific `tools` list enables only what it names, so repository MCP tools stay unavailable until you add `tracker/list_tickets` (or `tracker/*`). Instructions cannot grant tools (A). `tools: []` disables everything (C). Narrowing the server's `tools` array (D) does not help because the agent's list still excludes the server.

2.2: B. Cloud agent MCP configuration can reference only Agents secrets and variables whose names start with `COPILOT_MCP_`, using `$NAME`, `${NAME}` or `${NAME:-default}`. Actions and Codespaces secrets are not passed to the agent (A, D), `${{ secrets.X }}` belongs to custom agent YAML, and an unprefixed name (C) is not available to MCP servers.

2.3: B. `X-MCP-Toolsets` accepts a comma-separated list and `X-MCP-Readonly` restricts to read tools. The `/x/{toolset}` path takes a single toolset only (A). `/x/all` with lockdown (C) exposes everything and lockdown is only a content filter. A URL is not a stdio command (D).

2.4: B. `shell(git:*)` pre-approves Git commands, and deny rules override allows, removing `git push` and writes to `.env`. A grants all tools, paths and URLs. C misuses `--excluded-tools`, which takes tool names, and also allows everything. D is inverted.

2.5: C. The `managed-settings.json` allowlist is the generally available, recommended method, with secure matching by name, URL or `stdio` command that users cannot override. Registry-only enforcement (A, Preview) matches by name or ID and can be bypassed by editing configuration files. Disabling MCP (B) blocks approved servers too. Instructions (D) enforce nothing.

2.6: A and B. The job must be called `copilot-setup-steps`, and the workflow "won't trigger unless it's present on your default branch". The path `.github/workflows/copilot-setup-steps.yml` is already correct (D is wrong). Broad permissions (C) contradict least privilege and are not required. Separately, `timeout-minutes` is capped at 59.

2.7: B. The integrated firewall is not compatible with self-hosted runners and must be disabled, or the cloud agent is blocked; you then apply your own network controls. macOS is not supported (C). `container` is not among the honoured job keys (D). The recommended allowlist (A) belongs to the integrated firewall.

2.8: C. Specifying any permission sets the rest to `none`; the job needs `contents: read` and `copilot-requests: write` to make Copilot requests with `GITHUB_TOKEN`. B omits Copilot access. `copilot-requests` supports only `write` and `none` (D), and A is excessive.

2.9: C. Revert creates a new pull request that reverts the merge commit, preserving history and review. Deleting the source branch (A) or archiving the session (D) does not change `main`. Force-pushing (B) rewrites shared history and is usually blocked by protection rules.

2.10: A. Agentic events appear in the enterprise audit log with `actor:Copilot` (last 180 days), and include `agent_session_id` and the initiating `user`. The other sources lack attribution or coverage.

2.11: C. The cloud agent runs in an ephemeral GitHub Actions-powered environment, works on a `copilot/` branch from the selected base and returns a pull request, so the laptop can be off. The Local harness stops when the editor window closes (A); an interactive CLI session needs the machine (B); a tool set is not an execution context (D).

2.12: B. In `-p` mode, "any action that isn't pre-approved is denied automatically". The command pre-approved only ESLint, not `write`. Add `--allow-tool='write'` (or a narrower `write(...)` pattern). `-s` only trims output; `--no-ask-user` disables only the `ask_user` tool; the firewall concerns network traffic on the cloud agent.

Key takeaways

- Omit `tools` or use `["*"]` for all tools; a list enables only what it names; `[]` disables all. Unknown names are ignored.
- Aliases `read`, `edit`, `search`, `execute`, `agent`, `web`, `todo` make profiles portable; MCP tools are `server/tool` or `server/*`.
- Availability (tools lists, `--available-tools`) and permission (approvals, `--allow-tool`, `--deny-tool`) are separate layers; in Copilot CLI deny always wins, even over `--allow-all`.
- The cloud agent never asks before using a configured MCP tool, so allowlist specific, preferably read-only, tools.
- VS Code `.vscode/mcp.json` uses `servers` and `inputs`; `.mcp.json`, the CLI user file and the cloud agent repository JSON use `mcpServers`; cloud agent entries need `type` and `tools`.
- Cloud agent MCP secrets are Agents secrets prefixed `COPILOT_MCP_`; the agent cannot see Actions, Codespaces or Dependabot secrets.
- The cloud agent supports MCP tools only, and no OAuth-based remote servers; its default GitHub MCP token is read-only and scoped to the current repository.
- The remote GitHub MCP server lives at `https://api.githubcopilot.com/mcp/`; select toolsets with `/x/{toolset}` or `X-MCP-Toolsets`, add `/readonly` or `X-MCP-Readonly`, and authenticate with OAuth or `Authorization: Bearer` PAT.
- Custom registries and "Registry only" are Preview with bypassable name matching; `managed-settings.json` allowlists are GA and recommended.
- The cloud agent pushes only to its `copilot/` branch (or the pull request branch when invoked with `@copilot`), cannot approve or merge, and waits for **Approve and run workflows**.
- `copilot-setup-steps.yml`: single job `copilot-setup-steps`, on the default branch, honoured keys only, `timeout-minutes` max 59; self-hosted runners require disabling the firewall.
- For CI: `-p`, `-s`, `--no-ask-user`, narrow `--allow-tool`, `contents: read` plus `copilot-requests: write`; beware fork triggers.
- Bound retries, validate outputs, keep transcripts, escalate through issues and reviews, roll back with **Revert** or `git revert`, and trace with session logs, co-authored signed commits and `actor:Copilot` audit events.

References (Microsoft Learn, GitHub Docs and VS Code docs)

- Microsoft Learn: Tooling, MCP, and Agent Execution Environments
- Learn unit: How agents interact with GitHub APIs and workflows
- Learn unit: MCP servers, registries, and allow lists
- Learn unit: Execution context and boundaries
- Learn unit: Agent execution limits and protections
- GitHub Docs: Custom agents configuration
- GitHub Docs: Creating custom agents for Copilot cloud agent
- GitHub Docs: About custom agents (Copilot CLI)
- GitHub Docs: Custom agents and sub-agent orchestration (Copilot SDK)
- GitHub Docs: Configure MCP servers for your repository
- GitHub Docs: Model Context Protocol (MCP) and GitHub Copilot cloud agent
- GitHub Docs: Configure secrets and variables for Copilot cloud agent
- GitHub MCP Server: Remote GitHub MCP Server
- GitHub MCP Server: README (installation, authentication, default toolsets)
- GitHub Docs: MCP server usage in your company
- GitHub Docs: Configure an MCP registry for your organization or enterprise
- GitHub Docs: Restrict MCP server access to a custom registry
- GitHub Docs: MCP private registry enforcement
- GitHub Docs: Configuring an MCP server allowlist for your enterprise
- GitHub Docs: Allowing and denying tool use (Copilot CLI)
- GitHub Docs: Using Autopilot mode in GitHub Copilot CLI
- GitHub Docs: GitHub Copilot CLI command reference
- GitHub Docs: GitHub Copilot CLI programmatic reference
- GitHub Docs: GitHub Copilot CLI (trusted directories, allowed tools)
- GitHub Docs: Running GitHub Copilot CLI programmatically
- GitHub Docs: Automating tasks with Copilot CLI and GitHub Actions
- GitHub Docs: Using Copilot CLI in GitHub Actions with GITHUB_TOKEN
- GitHub Docs: Configure the development environment (copilot-setup-steps.yml)
- GitHub Docs: Customizing or disabling the firewall for GitHub Copilot
- GitHub Docs: Risks and mitigations for GitHub Copilot cloud agent
- GitHub Docs: Managing agent sessions
- GitHub Docs: Using Copilot cloud agent on GitHub
- GitHub Docs: Configuring settings for GitHub Copilot cloud agent
- GitHub Docs: Adding GitHub Copilot cloud agent to your organization
- GitHub Docs: Troubleshooting GitHub Copilot cloud agent
- GitHub Docs: Audit log events for agents
- GitHub Docs: Workflow syntax for GitHub Actions (permissions)
- GitHub Docs: GITHUB_TOKEN
- GitHub Docs: Reverting a pull request
- VS Code: MCP configuration reference
- VS Code: Add and manage MCP servers
- VS Code: Custom agents
- VS Code: Create and use tool sets
- VS Code: Use tools with agents
- VS Code: Tools and context reference
- VS Code: Manage approvals and permissions
- VS Code: Choose an agent harness
- VS Code: Review and revert agent changes

DOMAIN 3

Manage memory, state, and execution

Exam weight: 10–15%

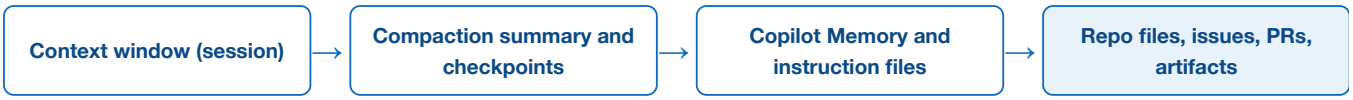
An agent is only as good as what it remembers and what it forgets. This domain tests whether you can decide *where* an agent's knowledge should live (in the context window, in Copilot Memory, in instruction files, or in GitHub artifacts such as commits, issues, pull requests and workflow artifacts), how long that knowledge should survive, and how to keep a long-running or multi-environment task consistent with the decisions already made.

Expect scenario questions that hinge on precise product behaviour: which Copilot CLI slash command reduces context versus discards it, how Copilot Memory validates and expires facts, how a stalled Copilot cloud agent session is resumed, how GitHub Actions passes data between jobs, and how competing instruction files are combined. Study it by doing: run a long Copilot CLI session and watch `/context`, assign an issue to Copilot cloud agent and steer it with `@copilot` comments, and build a two-job workflow that hands state over with job outputs and artifacts.

Official skill (Manage memory, state, and execution)	Bullet	Section
Implement agent memory strategies	Choose between short-term, long-term, and external memory	3.1.1
	Scope agent memory to task-relevant information	3.1.2
	Define memory expiration, pruning, and reset rules	3.1.3
Persist agent state and manage context drift	Capture task progress and decisions as durable artifacts	3.2.1
	Resume agent work without repeating steps or diverging from prior decisions	3.2.2
	Detect and correct drift during extended agent execution	3.2.3
Ensure continuity of agent memory and state across tools and environments	Share agent state	3.3.1
	Prevent conflicting context	3.3.2
	Prevent stale context	3.3.3

3.1 Implement agent memory strategies

A language model is stateless. Everything an agent "knows" during a request has to be assembled into the prompt: system instructions, customizations, your message, the conversation so far, implicit editor context, explicit references, and tool outputs. VS Code documents exactly these seven layers when it describes how it assembles context. Memory strategy is therefore the discipline of deciding which information enters that prompt, from which store, and for how long it stays eligible.



From most volatile (left) to most durable and auditable (right).

3.1.1 Choose between short-term, long-term, and external memory

Short-term memory is the context window of the current conversation. GitHub's Copilot CLI documentation describes it plainly: every prompt, every response, every tool call and its result, plus the system instructions and tool definitions, are held in a context window of fixed token size that varies by model. VS Code adds that conversation history is scoped to a single chat and is not automatically shared with other chats or sessions. Short-term memory is fast and needs no setup, but it is lost when the session ends or is cleared, and it degrades when it is summarised.

Long-term memory persists across sessions. In the GitHub ecosystem there are two main kinds:

- **Copilot Memory** PREVIEW: Copilot builds its own knowledge as it works. It stores *repository-level facts* (coding conventions, architectural decisions, build commands, project-specific rules) and *user-level preferences* (how a particular user likes to work with Copilot). It is currently used by Copilot cloud agent, Copilot code review, Copilot CLI, and agentic autofix, and a fact learned by one feature can be used by another. Feature limits matter: Copilot CLI applies repository-level facts plus the user-level preferences of the user who started the operation, while Copilot code review uses repository-level facts only.
- **Custom instructions**: human-written, version-controlled Markdown such as `.github/copilot-instructions.md`, path-specific `.github/instructions/**/*.instructions.md`, and `AGENTS.md` (or a single root `CLAUDE.md` or `GEMINI.md`). They are reviewed through pull requests like any other code and are loaded automatically. VS Code's guidance is explicit: if an instruction should apply across requests or sessions, store it in custom instructions instead of relying on conversation history.

VS Code also offers a local **memory tool** with three scopes: user memory (`/memories/`, persists across all workspaces, first 200 lines loaded into every session), repository memory (`/memories/repo/`, workspace-scoped), and session memory (`/memories/session/`, cleared when the conversation ends). These are stored on your machine only. VS Code's own advice is to promote stable, team-relevant repository memory into source-controlled instructions after verifying it against the code.

External memory is state held outside the agent runtime in systems that other people and tools can read: files committed to the repository (plans, progress logs, ADRs), issues and issue comments, pull request descriptions and review threads, commits, workflow job outputs and artifacts, and stores reachable through MCP servers. By default the GitHub MCP server that Copilot cloud agent uses can read context in the repository where the agent is working, such as issues and previous pull requests, and broader access can be configured. The Microsoft Learn module *Foundations of Agentic AI in GitHub* frames this as GitHub being the **system of record**: repositories, commits, pull requests, issues, workflow runs and artifacts, and review history make agent behaviour inspectable after the fact.

Memory type	Examples	Lifetime	Who can see it	Choose it when
Short-term	Copilot CLI or VS Code chat context window; compaction summary	Current session (survives `/resume` in the CLI, lost on `/clear`)	The user running the session (unless shared)	Exploratory work, a single focused task, iterating on a problem
Long-term, agent-curated	Copilot Memory repository facts and user preferences; VS Code memory tool	Copilot Memory: deleted after 28 days unused (timer can reset on validated use)	Repo facts: anyone with Copilot Memory access to that repo; preferences: that user	You want Copilot to learn conventions without hand-maintaining files
Long-term, human-curated	`.github/copilot-instructions.md`, `*.instructions.md`, `AGENTS.md`, organization and personal instructions	Until edited or deleted	Everyone with repo access; reviewed via PR	Rules that must apply every time and need review and audit

Memory type	Examples	Lifetime	Who can see it	Choose it when
External, durable	Plan and progress Markdown, issues, PR checklists, commits, Actions outputs and artifacts, MCP-backed stores	Repository history; artifacts 90 days by default	Collaborators, CI jobs, other agents	Multi-session, multi-agent, or multi-environment work that must be resumable and auditable

EXAM TIP

Map the question's wording to the store. "Retain details it has learned and use that knowledge for future work" means Copilot Memory. "Must always apply and be reviewed" means custom instructions. "Another job, another runner, another tool or another person must pick it up" means external memory (outputs, artifacts, repository files, issues and PRs). "Near the context limit in this session" means short-term memory management (`/compact`).

NOTE

Copilot Memory is enabled *per user*, not per repository. On individual plans it is on by default. On organization- or enterprise-managed plans it is off by default: an administrator enables the policy (organization **Settings** > **Copilot** > **Policies** > **Features** > **Copilot Memory**, or the enterprise **AI controls** page), after which it is on for licensed members, who can opt out in their personal settings. If several organizations grant a user a seat, the most restrictive setting applies: Memory is not used unless all of them enable it.

3.1.2 Scope agent memory to task-relevant information

Every token of irrelevant context competes with relevant context. GitHub's prompting best practices give three concrete scoping actions: in the IDE, open relevant files and close irrelevant ones; in Copilot Chat, delete a request from the conversation when it is no longer helpful; and if none of a conversation's context is helpful, start a new conversation. VS Code adds that explicit references still count toward the context window, so add only sources that help the current task, and keep each session focused on one task.

Scoping techniques you should recognise:

- **Session per task.** Copilot CLI's guidance is to start a fresh session when switching to an unrelated task, after many compactions, or when work went in a wrong direction. VS Code distinguishes a new chat in the same session (fresh history, shared workspace), a fork (inherits history up to the fork point), and a new session (fresh history, own workspace).
- **Explicit context.** In VS Code use `#`-mentions for files, folders, symbols, issues and pull requests rather than pasting large blobs.
- **Subagents for isolation.** A subagent works in its own context and returns only a result, keeping intermediate research out of the main window. In Copilot CLI a custom agent spawned as a subagent does not receive repository instruction files unless its frontmatter sets `include-custom-instructions: true`.
- **Large tool output offload.** Copilot CLI writes any tool output larger than 20 KiB to a temporary file and gives the model a path plus preview. `COPILOT_LARGE_OUTPUT_THRESHOLD_BYTES` changes the limit; raising it leaves less room for the conversation.
- **Path-scoped instructions.** Put rules that apply only to some files in `*.instructions.md` files with an `applyTo` glob, so they load only when the agent touches matching files. On GitHub.com, the `excludeAgent` key (`"code-review"` or `"cloud-agent"`) restricts which agent reads the file.
- **Repository-scoped memory.** Copilot Memory repository facts can be used only in operations on the same repository, which keeps what Copilot learns about one codebase out of others.

Listing 3.1: Path-specific instructions that load only for matching files (`.github/instructions/migrations.instructions.md`)

```
---
applyTo: "db/migrations/**/*.sql,src/**/repositories/**/*.ts"
excludeAgent: "code-review"
---
# Database migration rules

- Never edit a migration that has been merged to main; add a new one.
- Name files YYYYMMDDHHMM_short_description.sql.
- Every migration needs a matching down migration in the same PR.
- Run `npm run db:verify` before pushing.
- Record any schema decision in docs/agent/DECISIONS.md with the PR number.
```

In VS Code, an `.instructions.md` file may also carry `name` and `description` fields: VS Code attaches it automatically when `applyTo` matches a file being created or modified, and the agent can load it on demand when the `description` matches the task. A file with neither `applyTo` nor `description` must be attached manually.

COMMON TRAP

Switching from Ask to Agent, or to a "bigger" model, does not reduce context. The remedies for irrelevant context are removal (delete the request, close files, start a new chat or session) or summarisation (`/compact`), not a mode change. Deleting repository memories affects long-term facts, not the current chat.

3.1.3 Define memory expiration, pruning, and reset rules

Every memory layer needs an explicit rule for when information leaves. Know the built-in rules, then add your own team policy on top.

Copilot Memory. Repository facts are stored with citations to the code that supports them. To prevent stale information lingering, any fact or preference that goes unused is automatically deleted after **28 days**; the timer may reset whenever Copilot successfully validates and uses the entry. Repository owners and administrators can review and delete repository facts at repository **Settings** > **Copilot** > **Memory**. Users can view and delete their own preferences on any plan. On Copilot Business and Enterprise, organization or enterprise administrators can export (JSONL) or delete user-level preferences in bulk or per user for the billing entity they own; exports are reused for a two-hour window and limited to 10 per hour. Exports, deletions and user opt-outs appear in the audit log.

Copilot CLI context. When the conversation reaches roughly 80% of the context window, the CLI starts compacting in the background, leaving about 20% as buffer; if usage reaches about 95% before compaction finishes, it pauses briefly. Compaction replaces history with a structured summary (goals, work done, key technical details, important files, next steps) plus original user instructions and the current state of plans or to-do lists. Each compaction creates a numbered **checkpoint** that you can read with `/session checkpoints`. A completed compaction cannot be reversed, and fine detail (exact wording, full command output, minor early decisions) may be lost.

Listing 3.2: Copilot CLI commands for inspecting, pruning and resetting short-term memory

```
# How full is the window? Shows model, tokens used / capacity, and a breakdown:
# System Prompt, Custom Instructions, System Tools, MCP Tools, Messages, Free Space, Buffer
/context

# Summarise history now, steering what the summary keeps (Esc cancels a manual compaction)
/compact focus on the retry-policy decisions and the remaining TODO list

# Read what each compaction preserved
/session checkpoints
/session checkpoints 2

# Session metrics: per-model token totals (and AI credits on token-billed accounts)
/usage

# Roll back to an earlier user turn: conversation only, or conversation + files
/rewind
```

```
# Discard the conversation and start a new one (aliases: /new, /reset)
/clear

# Start an empty session in a new Git worktree, leaving this one untouched
/new worktree
```

Need	Copilot CLI	VS Code chat	Effect on memory
See context consumption	`/context`	Context window control in the chat input (hover for token breakdown)	None: read-only
Session cost and token totals	`/usage`	Hover the context window control (shows AI credits for the session)	None: read-only
Keep going in the same session with less history	`/compact [FOCUS-INSTRUCTIONS]` or automatic at about 80%	`/compact` or **Compact Conversation**; automatic unless `github.copilot.chat.summarizeAgentConversationHistory.enabled` is `false`	History replaced by a summary
Throw history away	`/clear`, `/new`, `/reset`	Start a new session or new chat	Nothing carried forward
Undo recent turns	`/undo` or `/rewind`	Restore a checkpoint	Rolls conversation (and optionally files) back
Clear persistent local memory	Not applicable	**Chat: Clear All Memory Files** (or ask the agent to update or delete one file)	User, repo and session memory files removed

External memory. Workflow artifacts are retained for 90 days by default; set a shorter `retention-days` on `actions/upload-artifact` (it cannot exceed the repository, organization or enterprise limit). Deleting a workflow run deletes its artifacts. Repository files are pruned by ordinary commits: delete or archive a finished plan, and remove instructions that no longer describe the code.

PRODUCTION NOTE

Write the rules down. A practical policy: session context is reset per task; plan and progress files are deleted (or moved to an archive folder) when the PR merges; instruction files are reviewed whenever the build, test, or architecture changes; Copilot Memory facts are reviewed by the repository owner after major refactors; state artifacts use a short `retention-days`. Microsoft's Zero Trust guidance on agentic memory adds that persistent memory is also a control plane: label provenance, isolate by user and tenant, and keep enough history to investigate and roll back poisoning.

3.2 Persist agent state and manage context drift

State is where a task stands: what has been decided, what is done, what remains, and what evidence proves it. **Context drift** is the gradual divergence between what the agent is doing and what was agreed, caused by summarised or truncated history, stale instructions, new and conflicting input, or a restart that lost the plan. The cure is the same in every

tool: externalise state into durable artifacts, resume from them deliberately, and check the agent against them while it runs.

3.2.1 Capture task progress and decisions as durable artifacts

A durable artifact is one that survives the session, is readable by humans and other agents, and is versioned. GitHub already provides several:

- **Commits.** Copilot cloud agent authors its commits as Copilot with the initiating user as co-author; each commit message links to the session logs, and the commits are signed and shown as Verified. A commit per completed step is the finest-grained progress record you can get.
- **Pull request description and checklists.** Copilot cloud agent works on one branch and opens one pull request per task, updating the PR as it works. A Markdown task list in the description is a shared, visible progress tracker.
- **Issue and PR comments.** Requirements, clarifications and review decisions live here, timestamped and attributed.
- **Session logs.** The session log shows the agent's reasoning and the tools it used. Logs are evidence, not a substitute for your own review.
- **Plan and progress Markdown files** committed to the branch. This is the most portable form and the one the exam scenarios describe.

Microsoft's GitHub Copilot upgrade agent for .NET is the canonical documented example of file-based agent state. It writes everything under `.github/upgrades/{scenarioId}/` (for example `dotnet-version-upgrade`):

File	Stage	Purpose
`assessment.md`	Assessment	Breaking changes, API compatibility problems, deprecated patterns, upgrade scope. You can add project context before planning.
`upgrade-options.md`	Assessment	Confirmed strategy decisions (bottom-up, top-down or all-at-once; in-place or side-by-side; package management and so on).
`plan.md`	Planning	Detailed specification: strategies, dependency paths, risk mitigations.
`scenario-instructions.md`	Planning	Preferences, key decisions and custom instructions for the scenario (sections: User Preferences, Key Decisions, Custom Instructions).
`tasks.md`	Execution	Sequential tasks with validation criteria; Copilot updates statuses and a progress figure as it works. The place to verify every step and the final test result.
`tasks/{taskId}/progress-details.md`	Execution	What the agent did for one task, in order, including errors.

You can apply the same pattern to any agent workflow. The template below is a team convention (GitHub does not mandate a file name); what matters is that it separates goals, decisions, and verifiable progress.

Listing 3.3: Progress file template committed to the working branch (`docs/agent/PROGRESS.md`)

```
# Task: Add idempotent retries to the payments client
Issue: #482 | Branch: copilot/payments-retry | PR: #497
Last updated: 2026-10-18 by @copilot (session link in latest commit)

## Goal and constraints
- Retries only for HTTP 429/503, max 3 attempts, exponential backoff with jitter.
- No new runtime dependencies. Public API of PaymentsClient must not change.

## Decisions (do not revisit without a new comment on the issue)
| # | Decision | Reason | Source |
|---|----------|--------|--------|
| D1 | Use the existing RetryPolicy helper in src/lib/retry.ts | Already tested, avoids a new package | Issue
#482 comment, 2026-10-16 |
| D2 | Idempotency key = request UUID header | Matches gateway contract | docs/adr/0012-idempotency.md |
```

```
## Progress
- [x] 1. Add RetryPolicy wiring to PaymentsClient (commit 3f2a9c1)
- [x] 2. Unit tests for 429/503 paths, `npm test -- payments` green
- [ ] 3. Integration test against sandbox gateway
- [ ] 4. Update docs/payments.md and CHANGELOG

## Validation evidence
- CI run: Actions run 912 (unit), artifact `payments-test-report`

## Next step
Start step 3. Do not change retry limits (D1) or header name (D2).
```

State to persist	Best home	Why
Requirements and acceptance criteria	Issue body and comments	Single, attributed source of intent; the cloud agent starts from the issue
Plan, decisions, progress checklist	Markdown file on the branch, mirrored as a PR checklist	Versioned with the code, survives sessions and machines, readable by any agent
Completed units of work	Commits (one per step)	Resumption skips completed steps; easy revert or cherry-pick
Review feedback and change requests	PR review comments (batched with **Start a review**)	Binds feedback to the PR session that will act on it
Small values between workflow jobs	Job outputs (`$GITHUB_OUTPUT`)	Typed strings, up to 1 MB per job
Files between jobs or after the run	Workflow artifacts	Cross-runner, retained (90 days default), digest-validated
Stable project conventions	Custom instructions	Loaded every time, reviewed via PR
Learned, citation-backed facts	Copilot Memory	Shared across Copilot features, self-expiring, validated before use

EXAM TIP

For the .NET upgrade agent, "verify each step and whether all tests passed" points to `tasks.md`; "which breaking changes were found" points to `assessment.md`; "the strategy and order" points to `plan.md`; "persist a preference or decision for the rest of the upgrade" points to `scenario-instructions.md`. All live in `.github/upgrades/{scenarioId}/`, not directly under `.github/`.

3.2.2 Resume agent work without repeating steps or diverging from prior decisions

Resumption has two halves: get back into the right session or branch, and re-establish the agreed state before the agent acts.

Copilot CLI. Sessions persist locally and can be resumed with their checkpoints. Name sessions so they are easy to find.

Listing 3.4: Resuming Copilot CLI sessions

```
# Start a named session
copilot -n payments-retry

# Later: resume by ID, ID prefix, or exact (case-insensitive) name
copilot --resume=payments-retry

# Resume the most recent session in this directory (conflicts with --resume)
copilot --continue

# Inside a running session: switch to another session from a picker
```

```
/resume
/session info
```

Copilot cloud agent. The documented recovery paths differ by situation, and the exam distinguishes them carefully:

Situation	Documented action	Result
PR shows "Copilot started work" but you cannot tell what is happening	Select **View session** in the PR timeline	Live-streamed session log
Agent going the wrong way while still running	Steer from the agents page prompt box below the session log	Follow-up applied after the current tool call (consumes AI credits)
More changes needed on an existing open PR, or guidance has changed	Comment on the PR mentioning `@copilot` (you need write access); batch several with **Start a review**	New session on the same branch; existing commits are kept
Session stuck and timed out (sessions time out after an hour; the hard execution limit is 59 minutes)	Unassign the issue from Copilot, then reassign it	Retry from the issue context
Stuck while responding to a comment	Add the same comment to the PR again	Comment re-processed
Task no longer needed	**Stop session**	Actions run ends; pushed commits preserved

Copilot only responds to comments from people with write access and only on open pull requests: once a PR is merged or closed, mentions are ignored. Mentioning `@github` is the IDE chat participant used to ask Copilot Chat to create a PR in VS Code, Visual Studio or JetBrains; it is not how you iterate on an agent PR.

The resume prompt. Whatever the tool, a resume prompt should point the agent at the durable state, forbid re-doing completed steps, and restate the decisions that must not change.

Listing 3.5: A resume prompt as a PR comment (or as the first prompt of a new CLI or VS Code session)

```
@copilot Resume this task from the current state of this branch.

1. Read docs/agent/PROGRESS.md and the last 5 commits before changing anything.
2. Steps 1–2 are complete and verified; do not redo or rewrite them.
3. Decisions D1 (use RetryPolicy helper) and D2 (UUID idempotency header) are final.
   If you think either is wrong, stop and explain in a comment instead of changing it.
4. Continue with step 3 (integration test). Run `npm test -- payments` before pushing.
5. Tick the checklist in PROGRESS.md and in this PR description as each step passes.
```

.NET upgrade agent. If `.github/upgrades/{scenarioId}` exists from a prior attempt, Copilot asks whether to continue or start fresh. If a task is stuck in progress, tell the agent "resume" or "restart the current task"; if that fails, "mark the current task as pending and restart it" or "reassess and continue from the last completed step", and check the task's `progress-details.md` to see where the previous session stopped. If the folder is deleted the agent cannot recover and must start fresh, which is why Microsoft recommends committing `.github/upgrades/` to the branch so state is preserved across sessions and machines.

> **COMMON TRAP**
> Do not "restart" a cloud agent task by creating a new issue or starting a new task from Copilot Chat when the goal is to continue the existing PR: that produces a second branch and PR and repeats work. Do not merge a draft PR to wake the agent; merged PRs are ignored. **Approve and run workflows** only lets CI run on Copilot's pushes; it does not resume the agent.

3.2.3 Detect and correct drift during extended agent execution

Drift is easier to prevent than to repair, but you must be able to spot it. Typical signals:

- The agent re-proposes an option that was rejected earlier, or rewrites code that a previous step completed.
- Changes stop matching repository conventions or the latest guidance in instructions or review comments.
- "Copilot seems to be forgetting earlier parts of the conversation", which GitHub lists as a reason to run `/context`.
- Tests that previously passed start failing, or progress files and commits disagree.
- Session logs show long loops with no progress, or the PR body status stops changing.

Detection and correction tools by surface:

Surface	Detect	Correct
Copilot CLI	`/context` to check usage and whether compaction is near; `/session checkpoints` to see what summaries kept; `/diff` to review changes	`/compact` with focus instructions that restate decisions; `/rewind` to roll back turns (optionally files); `/clear` and resume from the progress file when summaries are unreliable
VS Code	Context window control; References list in responses (which instructions were used); session changes	Delete an irrelevant request; restore a checkpoint; fork to explore an alternative; start a new session with the plan attached
Copilot cloud agent	**View session** logs; token usage and session length on the agents page; CI results on the PR	Steer the running session; `@copilot` review comments with corrected guidance; **Stop session** and restart with a better-scoped issue
.NET upgrade agent	`tasks.md` statuses; `progress-details.md`	"reload instructions" or start a new chat to reread `scenario-instructions.md`; `git revert HEAD` and ask for a different approach

Structural defences matter more than reactive ones. Copilot cloud agent sessions have a maximum execution time of 59 minutes that cannot be extended; GitHub's guidance is to break complex work into smaller, focused tasks, and you can only *shorten* the limit with `timeout-minutes` (maximum 59) in `copilot-setup-steps.yml`. Shorter tasks mean fewer compactions, less summarisation loss and smaller blast radius when drift happens.

Listing 3.6: Capping cloud agent run time in `.github/workflows/copilot-setup-steps.yml`

```
name: "Copilot Setup Steps"
on:
  workflow_dispatch:
  push:
    paths:
      - .github/workflows/copilot-setup-steps.yml

jobs:
  copilot-setup-steps:          # job name must be exactly this
    runs-on: ubuntu-latest
    timeout-minutes: 30         # maximum allowed value is 59
    permissions:
      contents: read
    steps:
      - uses: actions/checkout@v6
      - run: npm ci
```

EXAM TIP

Prefer the narrowest correction that preserves good work: steer or comment before stopping; `/compact` with focus before `/clear`; `/rewind` a few turns before abandoning the session. Choose a fresh start only when the scenario says the direction is wrong or the summaries have lost key decisions.

3.3 Ensure continuity of agent memory and state across tools and environments

Real agent work spans surfaces: an issue drafted on GitHub.com, a cloud agent PR, a local reproduction in Copilot CLI, a fix in VS Code, a CI pipeline with several jobs. Continuity means each surface sees the same, current truth.

3.3.1 Share agent state

Between GitHub Actions jobs. Every job runs on a fresh runner, so nothing in memory or on disk carries over. Environment variables written to `$GITHUB_ENV` reach only later steps of the *same* job, and the step summary (`$GITHUB_STEP_SUMMARY`) is a Markdown report, not a data channel. Use the two documented mechanisms:

- **Job outputs** for small values. A step writes `name=value` to `$GITHUB_OUTPUT` (the step needs an `id`); the job maps it under `jobs.<job_id>.outputs`; a downstream job declares `needs` and reads `needs.<job_id>.outputs.<name>`. Outputs are limited to 1 MB per job and 50 MB per workflow run (approximated as UTF-16). Outputs that contain secrets are redacted on the runner and skipped.
- **Artifacts** for files. `actions/upload-artifact` and `actions/download-artifact` move files between jobs, with `needs` enforcing order. In v4, artifacts are immutable (upload under a new name or use `overwrite: true`), hidden files are excluded unless `include-hidden-files: true`, and download validates the SHA256 `digest` output by the upload.

Listing 3.7: Passing an agent plan between jobs with outputs (small values) and an artifact (files)

```
name: agent-plan-then-implement
on: pull_request
permissions:
  contents: read

jobs:
  agent-plan:
    runs-on: ubuntu-latest
    outputs:
      plan-id: ${{ steps.plan.outputs.plan_id }}
      risk: ${{ steps.plan.outputs.risk }}
    steps:
      - uses: actions/checkout@v6
      - id: plan
        run: |
          mkdir -p agent-state
          ./scripts/generate-plan.sh > agent-state/plan.md
          echo "plan_id=$(sha256sum agent-state/plan.md | cut -c1-12)" >> "$GITHUB_OUTPUT"
          echo "risk=low" >> "$GITHUB_OUTPUT"
      - uses: actions/upload-artifact@v4
        with:
          name: agent-state
          path: agent-state/
          retention-days: 7
          if-no-files-found: error

  agent-implement:
    needs: agent-plan
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - uses: actions/download-artifact@v5
        with:
          name: agent-state
          path: agent-state
      - env:
          PLAN_ID: ${{ needs.agent-plan.outputs.plan-id }}
          RISK: ${{ needs.agent-plan.outputs.risk }}
        run: |
          echo "Implementing plan $PLAN_ID (risk: $RISK)"
          cat agent-state/plan.md
```

For a short multiline value (for example a one-paragraph plan summary) use the delimiter syntax from the workflow commands reference; for anything large or arbitrary, write a file and use an artifact instead, as GitHub advises.

Listing 3.8: Multiline step output with a delimiter

```
- id: summary
  run: |
    {
      echo "plan_summary<<EOF"
      head -n 20 agent-state/plan.md
      echo "EOF"
    } >> "$GITHUB_OUTPUT"
```

COMMON TRAP

`${{ steps.x.outputs.y }}` resolves only inside the job that ran step `x`; across jobs you need `${{ needs.job.outputs.y }}`. `${{ vars.* }}` are configuration variables, not runtime state. Caching is for reusable inputs such as dependencies, not for handing results to the next job: GitHub states that artifacts and caches cannot be used interchangeably.

Between Copilot surfaces. Several documented bridges exist:

- On GitHub.com, a cloud agent session started from Copilot Chat incorporates the chat's context, and Chat can answer questions about a Copilot PR by pulling in its session logs. This is scoped to the sessions you are working with and is distinct from Copilot Memory.
- Copilot Memory facts captured by one feature (for example cloud agent) are used by others (code review, CLI) in the same repository.
- Copilot CLI `--remote` with `--resume <TASK-ID>` resumes a remote task locally; remote control lets you monitor and steer a CLI session from GitHub.com while it runs on your machine.
- VS Code can discover local sessions from Copilot CLI and other supported apps (Copilot harness sessions tied to a repository and updated in the last seven days), and offers **Resume in Terminal** for Copilot harness sessions. A handoff carries the full conversation to another agent configuration, for example from the Plan agent to an implementation agent.
- Synced sessions from Copilot CLI, cloud agent and code review can be queried in natural language (only sessions you started; syncing must be allowed by policy).

These bridges are convenient but tool-specific. The bridge that works everywhere, for every agent and every human, is the repository and its issues and PRs. Treat them as the system of record and every other store as a cache.

3.3.2 Prevent conflicting context

Conflicts arise when two sources tell the agent different things: a personal instruction versus the repository file, a root `AGENTS.md` versus a nested one, an old chat message versus a new review comment, or a memory versus current code. Know the documented combination rules:

Sources in play	Documented behaviour
Personal, repository and organization instructions (Copilot on GitHub.com)	All relevant sets are provided. Personal instructions take the highest priority, then repository, then organization. GitHub advises avoiding conflicting sets.
`.github/copilot-instructions.md` plus a matching `*.instructions.md`	Both are used when the `applyTo` path matches the file being worked on.
Several `AGENTS.md` files	The nearest `AGENTS.md` in the directory tree takes precedence.
Copilot CLI, multiple user-level and repository files	Combined; identical duplicates removed; no general precedence order defined. Avoid conflicts.

Sources in play	Documented behaviour
VS Code, multiple instruction sources	Additive. Do not depend on order or precedence; resolve contradictions in the source files.
Copilot cloud agent with organization instructions	Uses organization instructions but prioritises repository-wide instructions first.

Practical rules for a single source of truth:

- **One owner per fact.** Build and test commands live in one file (for example `.github/copilot-instructions.md` or `AGENTS.md`); other files link to it. Copilot CLI supports `@path` imports in `AGENTS.md`, `CLAUDE.md` and `.github/copilot-instructions.md`, which avoids copy-paste divergence.
- **If several harnesses are used**, VS Code recommends a shared format such as `AGENTS.md`; where separate files are unavoidable, keep shared requirements consistent.
- **Decisions in one place.** Record decisions in the progress or ADR file and refer to them from prompts and PR comments, rather than restating them differently each time.
- **Inspect what is loaded.** In Copilot CLI, `/instructions` views and toggles instruction files, `copilot instruction list` lists sources non-interactively, and `--no-custom-instructions` disables them for a diagnostic run. In VS Code, expand **References** on a response and use the Agent Customizations editor and Agent Debug Logs. On GitHub.com, check whether `.github/copilot-instructions.md` appears in the response references.
- **Batch feedback.** Because the cloud agent starts acting on a comment as soon as it is submitted, use **Start a review** so it receives all related comments at once instead of reacting to partial, possibly contradictory, ones.

> **NOTE**
> VS Code's documented procedure for a conflict (for example organization rules require an approved HTTP client, a personal rule asks for another library) is: reproduce in a new chat, inspect discovered sources and References, then fix the contradiction at the source by removing or narrowing the personal rule and aligning repository and targeted instructions. Adding yet another instruction to "win" is explicitly discouraged.

3.3.3 Prevent stale context

Stale context is correct information that has stopped being correct: a renamed build script still described in instructions, a memory about a module that was deleted, a plan for an approach the team abandoned.

How Copilot Memory handles staleness. When Copilot finds a relevant repository fact, it checks the fact's citations against the *current branch* and uses only validated facts. This also covers facts captured from pull requests closed without merging: they do not affect behaviour unless the current codebase still substantiates them. User-level preferences are checked with Copilot's judgment rather than code citations. Unused entries expire after 28 days. Note the distinction: code that no longer supports a fact makes it fail validation (so it is not used), but deletion happens through expiry or a manual delete.

How you handle staleness elsewhere:

- **Refresh instructions on change.** Update or remove instructions in the same PR that changes the build, test or architecture they describe. Re-run `/init` (CLI) or regenerate from the Agent Customizations editor, then review: VS Code warns that generated commands and conventions might be incomplete.
- **Verify instructions.** VS Code's procedure: confirm the file is discovered, start a new chat, run a small task with a clear success criterion, check References, and review the result and tool activity.
- **Force a reread.** Long-running sessions may not reread edited files. The .NET upgrade troubleshooting guide says to ask the agent to "reload instructions" or start a new chat session so changes to `scenario-instructions.md` take effect. For the cloud agent, a new `@copilot` comment starts a fresh session that picks up the branch's current instructions.
- **Expire external state.** Short `retention-days` for state artifacts; delete or archive progress files when the PR merges; close obsolete issues so agents reading issue context through MCP do not act on them.

- **Prune memory.** Repository owners review Copilot Memory facts after refactors; VS Code users promote stable repository memory into instructions and then update or remove the local copy; **Chat: Clear All Memory Files** resets local memory.
- **Refresh web context.** VS Code caches fetched pages for a limited time; restart VS Code to force a refresh.

PRODUCTION NOTE

Treat any persisted memory as untrusted input that can carry yesterday's mistakes or an injected instruction forward. Microsoft's guidance for agentic memory safety recommends gating memory writes on intent and provenance, never storing credentials or secrets, isolating memory by user and tenant with deterministic controls rather than prompting, and retaining enough history for incident investigation and rollback. Copilot Memory's design reflects several of these: repository facts are created only from activity by users with write access, are scoped to one repository, carry citations, and are auditable and deletable by owners.

Context window	The fixed-size, model-dependent token budget holding instructions, tool definitions, messages and tool results for the current conversation.
Compaction	Summarising conversation history to free context; automatic in Copilot CLI at about 80% usage and in VS Code when the window fills; manual with `/compact`. Not reversible.
Checkpoint (Copilot CLI)	A numbered, titled copy of a compaction summary stored in the session workspace; viewed with `/session checkpoints [n]`.
Copilot Memory	Public preview feature that stores citation-backed repository facts and user preferences, validates them before use, and deletes entries unused for 28 days.
Billing entity	The organization or enterprise that grants the licence and owns a user's Copilot Memory preferences; users with several licences must choose a default one to generate preferences.
Durable artifact	Versioned, shareable state outside the agent runtime: commits, plan and progress files, issue and PR comments, PR checklists, workflow artifacts.
Context drift	Divergence of an agent's behaviour from agreed goals and decisions over a long or interrupted execution.
System of record	GitHub's role in agent workflows: repositories, commits, PRs, issues, workflow runs, artifacts and reviews record intent, actions, evidence and decisions.
Job output	A string value set via `$GITHUB_OUTPUT`, mapped in `jobs.<job_id>.outputs` and read by dependent jobs through `needs`; up to 1 MB per job.

3.4 Additional practice questions

Practice 3.1 MULTIPLE CHOICE

Fabrikam uses Copilot Business with Copilot Memory enabled. Developer Ana often tells Copilot CLI that she prefers table-driven unit tests, and Copilot CLI now applies that preference. When Copilot code review reviews Ana's pull requests, it never mentions table-driven tests. You need to explain the behaviour. What is the reason?

A. User-level preferences expire after seven days unless Ana confirms them.
B. Copilot code review applies repository-level facts only, not user-level preferences.
C. Copilot Memory must be enabled separately in each repository's code review settings.
D. User-level preferences are available only in VS Code and Copilot CLI.

Practice 3.2 MULTIPLE CHOICE

A contractor receives Copilot seats from two organizations. Organization A has enabled the Copilot Memory policy; Organization B has not. You need to predict whether Copilot Memory will be used for the contractor. What happens?

A. Copilot Memory is used only in repositories owned by Organization A.
B. Copilot Memory is used everywhere because at least one organization enabled it.
C. Copilot Memory is not used, because the most restrictive setting applies.
D. Copilot Memory is used after the contractor opts in through each repository's settings.

Practice 3.3 MULTIPLE CHOICE

A workflow has an `analyse` job that produces a 4 MB JSON dependency map. A later `agent-fix` job, running on a different hosted runner, must read the complete map. You need a supported mechanism that does not modify the repository. What should you use?

A. Write the JSON to `$GITHUB_OUTPUT` and read it with `needs.analyse.outputs`.
B. Write the JSON to `$GITHUB_ENV` in the `analyse` job.
C. Upload the file with `actions/upload-artifact` and download it in `agent-fix`, which declares `needs: analyse`.
D. Append the JSON to `$GITHUB_STEP_SUMMARY` and parse the summary in `agent-fix`.

Practice 3.4 MULTIPLE CHOICE

You have run a Copilot CLI session for several hours and it has compacted automatically twice. The agent now suggests a caching library that the team rejected early in the session. Before deciding whether to continue or start fresh, you need to see what the earlier summaries preserved. Which command should you run?

A. `/usage`
B. `/session checkpoints`
C. `/context`
D. `/diff`

Practice 3.5 MULTIPLE SELECT (CHOOSE TWO)

A developer's personal custom instructions say "Always add semicolons". The repository's `.github/copilot-instructions.md` says "Follow StandardJS: no semicolons". The developer uses Copilot Chat on GitHub.com with the repository attached. Which two statements describe the documented behaviour?

A. Personal instructions take the highest priority.
B. Only the repository instructions are sent, because the repository is attached.
C. All relevant sets of instructions are provided to Copilot.
D. Organization instructions, if present, override both.
E. The most recently edited instruction file wins.

Practice 3.6 MULTIPLE CHOICE

A monorepo has an `AGENTS.md` at the root and another at `services/billing/AGENTS.md`. Copilot cloud agent is editing `services/billing/src/invoice.ts`. You need to know which agent instructions take precedence. Which file?

A. The root `AGENTS.md`, because repository-level files always win.
B. `services/billing/AGENTS.md`, because the nearest file in the directory tree takes precedence.
C. Neither, unless `.github/copilot-instructions.md` imports them.
D. Whichever file was committed most recently.

Practice 3.7 MULTIPLE CHOICE

You started a .NET 6 to .NET 10 upgrade with GitHub Copilot upgrade on a laptop. Assessment and planning are complete and two execution tasks are done. Tomorrow you must continue on a workstation that has a fresh clone of the repository. You need the agent to resume without repeating completed tasks or re-deciding the upgrade strategy. What should you do today?

A. Export the chat conversation and import it on the workstation.
B. Commit the `.github/upgrades/` folder to the working branch and push it.
C. Copy only `assessment.md` to the workstation.
D. Nothing; on the workstation, ask the agent to "reassess and re-plan".

Practice 3.8 MULTIPLE CHOICE

Issues describing large, multi-module migrations are assigned to Copilot cloud agent and regularly time out before finishing. A teammate proposes setting `timeout-minutes: 120` in `copilot-setup-steps.yml`. You need a solution that lets the work complete. What should you do?

A. Apply the teammate's change; the agent then runs for up to 120 minutes.
B. Switch the setup job to a larger runner so the session limit increases.
C. Split each migration into smaller, well-scoped issues, each completable within the session limit.
D. Mention `@copilot extend` on the PR before the session times out.

Answers and explanations

3.1: B. GitHub documents that Copilot code review uses repository-level facts only; user-level preferences are not applied during code review. Copilot CLI applies repository facts plus the initiating user's preferences, which is why Ana sees the behaviour there. Retention is 28 days of non-use, not seven (A); Memory is enabled per user by policy, not per repository (C); preferences are not limited to IDE and CLI surfaces (D).

3.2: C. When a user is assigned Copilot by more than one organization, the most restrictive setting applies: Memory is not used unless all of those organizations enable it. There is no per-repository opt-in (D), and enablement is per user, not per owning organization (A).

3.3: C. Job outputs are limited to 1 MB per job, so 4 MB rules out A. `$GITHUB_ENV` reaches only later steps in the same job (B), and the step summary is a report capped at 1 MiB per step, not a data channel (D). Artifacts move files across runners, with `needs` enforcing ordering and the download validating the digest.

3.4: B. Each compaction creates a checkpoint containing the summary; `/session checkpoints` lists them and `/session checkpoints N` shows one, which reveals whether the rejection decision survived. `/context` shows token usage, `/usage` shows session metrics, and `/diff` shows file changes.

3.5: A, C. Personal instructions have the highest priority, then repository, then organization, but all relevant sets are still provided to Copilot, which is why GitHub advises avoiding conflicts. Organization instructions have the lowest priority (D), and there is no last-edited rule (E).

3.6: B. GitHub documents that the nearest `AGENTS.md` in the directory tree takes precedence. `AGENTS.md` files are discovered without needing to be imported (C), and commit time is irrelevant (D).

3.7: B. The upgrade agent stores all state in `.github/upgrades/`; Microsoft recommends committing that folder so it is preserved across sessions and machines. When the folder exists, the agent offers to continue. Re-planning (D) regenerates the state and risks different decisions, and `assessment.md` alone (C) lacks the plan, options and task status.

3.8: C. Each cloud agent session has a maximum execution time of 59 minutes that cannot be extended; `timeout-minutes` in `copilot-setup-steps.yml` accepts a maximum of 59 and can only shorten the limit (A). Runner size does not change the limit (B), and there is no extend command (D). GitHub's guidance is to break complex work into smaller, focused tasks.

Key takeaways

- Short-term memory is the context window; long-term memory is Copilot Memory (agent-curated, preview) and custom instructions (human-curated); external memory is the repository, issues, PRs, commits, workflow outputs and artifacts, and MCP-reachable stores.
- Copilot Memory: per-user enablement (on by default for individual plans; admin policy first for organization and enterprise seats, then users can opt out); repository facts are repository-scoped, citation-backed and validated against the current branch; unused entries are deleted after 28 days; repository owners delete facts under repository Settings > Copilot > Memory.
- Copilot CLI: `/context` shows usage, `/usage` shows metrics, `/compact` summarises (auto at about 80%), `/clear` = `/new` = `/reset` discards, `/session checkpoints` shows summaries, `/rewind` rolls back, `--resume` and `--continue` resume.
- Persist state as plan and progress Markdown on the branch, PR checklists, one commit per step, and issue or PR comments; the .NET upgrade agent's `.github/upgrades/{scenarioId}/tasks.md` is the documented example of a live progress record.
- Cloud agent recovery: **View session** to inspect, steer while running, `@copilot` on the open PR (write access required) to continue on the same branch, unassign and reassign the issue to retry after a timeout; sessions are capped at 59 minutes, so scope tasks small.
- Between jobs: `$GITHUB_OUTPUT` + `jobs.<id>.outputs` + `needs.<id>.outputs` for small values (1 MB per job), artifacts for files (90-day default retention, immutable in v4).
- Instruction conflicts: on GitHub.com personal > repository > organization, all provided; nearest `AGENTS.md` wins; CLI and VS Code combine sources without a general precedence, so fix contradictions at the source.
- Stale context is prevented by validation (Memory citations), expiry (28 days, artifact retention), refresh (new session, "reload instructions", `@copilot` comment) and pruning outdated instructions in the same PR that changes the code.

References

- Study guide for Exam GH-600: Developing in Agentic AI Systems (Microsoft Learn)
- Describe GitHub as the system of record and control plane (Microsoft Learn training)
- Upgrade a .NET app with GitHub Copilot upgrade (Microsoft Learn)
- Troubleshoot GitHub Copilot upgrade (Microsoft Learn)
- Manage memory safety in agentic systems (Microsoft Learn)
- About GitHub Copilot Memory
- Managing Copilot Memory for an organization or enterprise
- Managing Copilot Memory for your personal account
- GitHub Copilot CLI command reference
- Managing context in GitHub Copilot CLI
- Adding custom instructions for GitHub Copilot CLI
- Adding repository custom instructions for GitHub Copilot
- Support for different types of custom instructions
- Best practices for using GitHub Copilot
- About GitHub Copilot cloud agent
- Troubleshooting GitHub Copilot cloud agent
- Managing agent sessions
- Best practices for using GitHub Copilot to work on tasks
- Configure the development environment (copilot-setup-steps.yml)
- Passing information between jobs
- Workflow syntax for GitHub Actions
- Workflow commands for GitHub Actions
- Store and share data with workflow artifacts
- Workflow artifacts
- actions/upload-artifact README

- Understand context in AI agents (VS Code)
- Add context to chat (VS Code)
- Manage agent sessions in VS Code
- Understand sessions and chats (VS Code)
- Understand AI agents (VS Code)
- Use memory with agents in VS Code
- Use custom instructions in VS Code

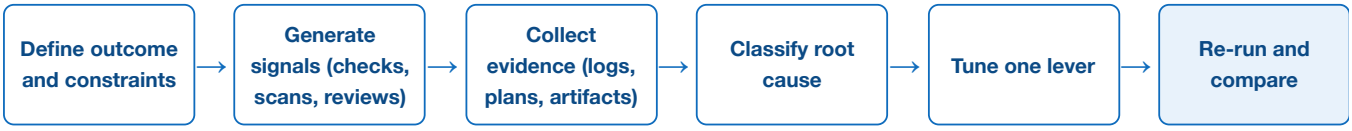
DOMAIN 4

Perform evaluation, error analysis, and tuning

Exam weight: 15–20%

This domain tests whether you can treat an AI agent like any other contributor whose work must be measured, debugged and improved. You are expected to decide what "done" means for an agent task before it starts, choose evidence that proves (or disproves) success, read the traces an agent leaves behind, put a failure into the right root-cause class, and then change the right lever: instructions, workflows, guardrails, memory or tool access.

Expect scenario questions built on a session log excerpt, a firewall warning, a failing check or a misbehaving custom agent, asking what the evidence proves, where to look next, or which file or setting to change. Practise the loop below on a real repository.



Official skill (Perform evaluation, error analysis, and tuning)	Bullet	Section
Define success criteria and evaluation signals for agent tasks	Specify expected outcomes and operational constraints for agent tasks	4.1.1
	Identify qualitative and quantitative evaluation signals to evaluate agents	4.1.2
	Align evaluation criteria with development intent	4.1.3
	Generate evaluation signals by using automated scanning tools	4.1.4
Analyze agent failures and identify root causes	Identify failures by using logs, plans, traces, outputs, and workflow artifacts	4.2.1
	Classify root causes, including reasoning errors, tool misuse, and context or environment issues	4.2.2
Tune agent behavior based on evaluation results	Revise instructions, workflows, or constraints	4.3.1
	Refine memory usage	4.3.2
	Refine tool usage and tool access	4.3.3

4.1 Define success criteria and evaluation signals for agent tasks

An agent cannot be evaluated against an intent that was never written down. Turn vague requests into a task contract with an observable outcome, explicit limits, and machine-generated evidence that the outcome was reached without collateral damage.

4.1.1 Specify expected outcomes and operational constraints for agent tasks

GitHub's guidance for Copilot cloud agent (formerly Copilot coding agent) is that it produces better results on clear, well-scoped tasks. An ideal task includes three things: a clear description of the problem or work, **complete acceptance criteria** describing what a good solution looks like (for example, whether unit tests are required), and directions about which files need to change. When you assign an issue, the issue body effectively *is* the prompt, so write it as one.

The implementation planner example in the GitHub customization library makes the same point from the planning side: its plan template starts with an Overview that must state "Success criteria (what does 'done' look like?)", followed by technical approach, phased tasks with complexity estimates, and a Considerations section that lists assumptions, **constraints** and risks, plus a "Not Included" list. A plan that lacks success criteria or out-of-scope items is not ready to hand to an implementer.

Element of the task contract	What to write	Why it matters for evaluation
Outcome	The observable behaviour change, for example "GET /orders returns 404 for unknown IDs".	Gives a pass/fail test the reviewer and CI can check.
Acceptance criteria	Tests that must exist and pass, docs to update, performance or security expectations.	Converts intent into required checks and review points.
Scope	Files or directories to change, and explicitly what is out of scope.	Makes scope creep detectable in the diff.
Operational constraints	Do not modify CI workflows, do not weaken or delete tests, no new dependencies without justification, keep the public API stable.	Defines the rules an otherwise "green" PR can still break.
Environment limits	Runtime versions, available services, network restrictions, time budget.	Separates environment failures from reasoning failures later.

Platform constraints also shape the contract. Each Copilot cloud agent session has a maximum execution time of 59 minutes that cannot be extended (`timeout-minutes` in `copilot-setup-steps` can only shorten it, maximum `59`). The agent changes only the repository the task started in, works on one branch, opens one pull request per task, reaches the internet through a firewall, cannot approve or merge its own PRs, and its pushes do not trigger workflows until someone with write access clicks **Approve and run workflows** (unless an administrator turns that requirement off).

Listing 4.1: An issue body written as a task contract for Copilot cloud agent (Markdown)

```
## Problem
`GET /api/orders/{id}` returns HTTP 500 when the order does not exist.

## Expected outcome
- Unknown IDs return HTTP 404 with body `{"error": "order_not_found"}`.
- Existing behaviour for valid IDs is unchanged.

## Acceptance criteria
- Add unit tests in `tests/api/test_orders.py` for the 404 path.
- `npm test` and `npm run lint` pass locally in your environment.
- No new runtime dependencies.

## Scope and constraints
- Change only `src/api/orders/**` and `tests/api/**`.
- Do NOT modify files under `.github/workflows/`.
- Do NOT delete, skip or loosen existing tests or assertions to make checks pass.
- If the fix needs a schema change, stop and explain why in the PR description.

## Out of scope
- Refactoring the order repository layer.
```

EXAM TIP

When a question asks how to "define success criteria" for an agent, prefer answers that make success *observable and enforceable* (acceptance criteria plus required checks) over answers that only add adjectives to the prompt ("be careful", "be accurate"). GitHub explicitly lists vague quality directives as instructions that add noise for Copilot code review.

4.1.2 Identify qualitative and quantitative evaluation signals to evaluate agents

No single signal tells you whether an agent is doing good work. A green build can hide a deleted test; a merged PR can hide three rounds of correction. Combine **quantitative** signals (counts, rates, durations, pass/fail states) with **qualitative** signals (review comments, session-log reasoning, human feedback) and always note what each signal cannot prove.

Signal	Type	Where you find it	What it shows	What it does not prove
Test pass rate / required check status	Quantitative	Checks tab and merge box of the PR; Actions run logs	The change satisfies the automated tests that ran	That tests are meaningful or unchanged; that the checks actually ran (workflows wait for approval on agent pushes)
Code scanning, secret scanning and dependency alerts	Quantitative	Security and quality tab, PR annotations, rulesets	New vulnerabilities, leaked credentials, vulnerable or disallowed dependencies	Functional correctness or design quality
Review comments (human and Copilot code review)	Qualitative (countable)	PR conversation and Files changed	Design problems, missed edge cases, convention breaks	Absence of comments does not prove absence of defects
Iteration count	Quantitative	Number of `@copilot` follow-up comments, steering messages and agent commits on the PR	How much human correction the agent needed	Why it needed correction (read the session log)
PR acceptance / merge rate, time to merge	Quantitative	Copilot usage metrics (pull request lifecycle metrics); repository PR history	Throughput and cycle time of agent-created PRs compared with all PRs	Quality of what was merged; causes of delay
Session length and token usage	Quantitative	Session log overview in the agents panel; VS Code Agent Debug Logs Summary view	Cost and efficiency; candidates for setup-step or instruction tuning	Correctness
Session-log reasoning and tool calls	Qualitative	Agents panel session log; VS Code Agent Debug Logs and Chat Debug view	What the agent believed, which tools it used, what validation it ran	Logs do not replace your own review and testing

Copilot usage metrics (APIs, a 28-day dashboard, code generation and impact dashboards, NDJSON export) are the documented source for organisation and enterprise-level trends. The metric categories are adoption, engagement, acceptance rate, lines of code, and **pull request lifecycle metrics** (pull request creation and merge counts, median time to merge, and review suggestion activity). By comparing all PR activity with PRs created by Copilot you can judge how agent work affects throughput and cycle time. Two practical caveats: data is available within two full UTC days, and most metrics come from client-side IDE telemetry, so end users need telemetry enabled for the richest data. Copilot code review adoption is reported separately with distinct active and passive user counts.

> **NOTE**
> Session-level evidence lives with the agent. On GitHub.com you open a session from the agents panel or the **View session** button in the PR timeline; the overview shows progress, token usage and session length. Commits from Copilot cloud agent are authored by Copilot with the person who started the task as co-author, are signed ("Verified"), and include a link to the session logs in the commit message, so a reviewer or auditor can trace any change back to its reasoning.

4.1.3 Align evaluation criteria with development intent

Any measurable target invites shortcuts. An agent rewarded only for "checks are green" can reach that state by deleting a failing test, adding a skip marker, lowering a coverage threshold, or editing `.github/workflows/` so the failing job no longer runs. Each of these satisfies the metric and defeats the intent. Evaluation criteria must therefore check *how* the

signal turned green, not only *that* it did.

- **State the intent as a constraint** in the task and in repository instructions: tests and CI configuration are part of the specification, not obstacles.
- **Make protected files require a different approver.** GitHub's guardrail tutorial recommends protecting important Copilot and MCP configuration files with a `CODEOWNERS` file and enabling the "Require review from Code Owners" rule, so edits must be approved by specific teams. Apply the same pattern to `.github/workflows/` and to test helpers.
- **Do not auto-run workflows on agent pushes without review.** By default, workflows do not run until a user with write access approves them. GitHub warns you to be especially alert to proposed changes in `.github/workflows/`, because workflows can be privileged and access secrets.
- **Pin the source of required checks** by selecting the expected GitHub App in the "Require status checks to pass" rule.
- **Count independent approvals.** The person who asked Copilot to create a PR cannot approve it; PRs Copilot opens under its own app identity need one extra approval by default.

Listing 4.2: CODEOWNERS entries that route CI, test and agent-configuration changes to a platform team

```
# .github/CODEOWNERS  (last matching pattern wins)
*                                 @contoso/app-devs

# CI definitions and the agent's own environment
/.github/workflows/               @contoso/platform-eng
/.github/hooks/                   @contoso/platform-eng

# Agent behaviour: instructions and custom agents
/.github/copilot-instructions.md  @contoso/platform-eng
/.github/instructions/            @contoso/platform-eng
/.github/agents/                  @contoso/platform-eng
/AGENTS.md                        @contoso/platform-eng

# Shared test infrastructure
/tests/fixtures/                  @contoso/qa-leads
```

CODEOWNERS on its own only requests reviews. It becomes a *gate* when a branch ruleset (or branch protection) requires a pull request with code owner review. Listing 4.3 shows a branch ruleset created through the REST API (`POST /repos/{owner}/{repo}/rulesets`) that combines the three controls you will see most often in exam scenarios: required pull request with code owner review, required status checks, and required code scanning results.

Listing 4.3: Branch ruleset (REST API body) enforcing code owner review, required checks and CodeQL results on the default branch

```
{
  "name": "default-branch-agent-guardrails",
  "target": "branch",
  "enforcement": "active",
  "conditions": {
    "ref_name": { "include": ["~DEFAULT_BRANCH"], "exclude": [] }
  },
  "rules": [
    {
      "type": "pull_request",
      "parameters": {
        "required_approving_review_count": 1,
        "require_code_owner_review": true,
        "dismiss_stale_reviews_on_push": true,
        "require_last_push_approval": true,
        "required_review_thread_resolution": true
      }
    },
    {
      "type": "required_status_checks",
      "parameters": {
        "strict_required_status_checks_policy": true,
        "required_status_checks": [
```

```
          { "context": "unit-tests" },
          { "context": "lint" },
          { "context": "dependency-review" }
        ]
      }
    },
    {
      "type": "code_scanning",
      "parameters": {
        "code_scanning_tools": [
          {
            "tool": "CodeQL",
            "alerts_threshold": "errors",
            "security_alerts_threshold": "high_or_higher"
          }
        ]
      }
    }
  ]
}
```

COMMON TRAP

A ruleset does not stop an agent from *proposing* a workflow edit; it stops that edit from being *merged* without the right review and checks. If you need to block the push itself, use a push ruleset with "Restrict file paths" (available for private and internal repositories and their fork networks) or a cloud agent `preToolUse` hook (section 4.3.1). Disabling CI for agent PRs, or relying on `.gitignore`, is never a control.

PRODUCTION NOTE

Rulesets apply to Copilot cloud agent just as they apply to humans. A rule that the agent cannot satisfy, for example one restricting commit authors, blocks the agent entirely; repository administrators can add Copilot as a bypass actor for that ruleset. Review existing rules before rolling out the agent, and target agent-enabled repositories with a custom property.

4.1.4 Generate evaluation signals by using automated scanning tools

Automated scanners turn security and quality expectations into repeatable, comparable signals. You meet them at two layers.

Inside the agent session. By default, Copilot cloud agent checks the code it generates and attempts to fix issues before completing the pull request: **CodeQL** for code security issues, the **GitHub Advisory Database** for newly introduced dependencies (malware advisories and CVSS High or Critical vulnerabilities), **secret scanning** for API keys, tokens and other secrets, and a second opinion from **Copilot code review**. This validation does not require a GitHub Secret Protection, GitHub Code Security or GitHub Advanced Security licence, and its results appear in the session log. A repository administrator can toggle each tool under **Settings > Copilot > Cloud agent > Validation tools**, for example to speed the agent up or avoid conflicts with other products.

On the pull request and default branch. These signals are independent of the agent and are what you should gate merges on:

Tool	Signal produced	How to enforce	Notes
CodeQL code scanning, default setup	Code scanning alerts on PRs and branches	Ruleset rule "Require code scanning results" with alert and security-alert thresholds	Eligible when GitHub Actions is enabled and the repository is public or has GitHub Code Security. Enable under Settings > Advanced Security > CodeQL analysis > Set up > Default.

Tool	Signal produced	How to enforce	Notes
CodeQL code scanning, advanced setup	Same, from your own workflow (`github/codeql-action/init` and `analyze`)	Same ruleset rule, or the job as a required status check	Switching to default setup disables the existing workflow and blocks CodeQL API uploads, so do not run both.
Secret scanning and push protection	Alerts for committed credentials; blocked pushes	Push protection; ruleset rule "Require secret scanning alerts are resolved" PREVIEW	Push protection for repositories requires GitHub Secret Protection.
Dependency review action	Fails the PR when added dependencies have vulnerabilities (by `fail-on-severity`) or disallowed licences	Required status check	`actions/dependency-review-action@v4`
Dependabot alerts	Alerts on the default branch when new advisories match your dependency graph	Triage SLAs; security updates	A lagging signal for merged agent work.
Unit tests, linters, type checks	Pass/fail check runs	Required status checks (strict policy keeps branches up to date)	Name jobs stably; the job's check name is what the ruleset requires.

Listing 4.4: Pull request workflow that produces test, lint, CodeQL and dependency review signals for use as required checks

```
name: pr-quality-gates
on:
  pull_request:
    branches: [main]

permissions:
  contents: read

jobs:
  unit-tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - uses: actions/setup-node@v7
        with:
          node-version: 22
          cache: npm
      - run: npm ci
      - name: Run unit tests
        run: npm test

  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - uses: actions/setup-node@v7
        with:
          node-version: 22
          cache: npm
      - run: npm ci
      - run: npm run lint

  codeql:
    runs-on: ubuntu-latest
    permissions:
      actions: read
      contents: read
      security-events: write
    steps:
```

```
      - uses: actions/checkout@v6
      - uses: github/codeql-action/init@v4
        with:
          languages: javascript-typescript
          build-mode: none
      - uses: github/codeql-action/analyze@v4

  dependency-review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - uses: actions/dependency-review-action@v4
        with:
          fail-on-severity: high
```

`init` prepares the CodeQL database and `analyze` runs the queries and uploads results. `build-mode: none` suits interpreted languages (and C/C++, C#, Java and Rust); Go, Kotlin and Swift need `autobuild` or `manual`. Only the CodeQL job gets `security-events: write`. Register `unit-tests`, `lint` and `dependency-review` as required status checks, and gate CodeQL through "Require code scanning results" so the severity threshold, not just job success, decides.

COMMON TRAP

A required check that never ran is not a pass. Because agent pushes wait for **Approve and run workflows**, an agent PR can sit with pending checks. Required status checks keep it unmergeable until they run and pass, which is exactly the behaviour you want; do not "fix" it by removing the requirement.

4.2 Analyze agent failures and identify root causes

Error analysis is evidence-led: establish what the agent did, in what order, with which tools and context, then name the root cause class, because each class has a different fix. Rewriting instructions to fix a firewall block wastes iterations and hides the real defect.

4.2.1 Identify failures by using logs, plans, traces, outputs, and workflow artifacts

Each surface leaves different evidence. Know where each type lives and what question it answers.

Evidence	Where (exact UI or command)	Answers the question
Cloud agent session log	Agents panel or `github.com/copilot/agents`; **View session** on the "Copilot started work" PR timeline event; link in each agent commit message	What did Copilot reason, which tools did it call, what tests and linters did it run, what did the built-in security validation find? Logs stream live while the session runs.
Underlying Actions job logs for the session	In the session log viewer, the ellipsis (**...**) menu, then **Copilot** in the sidebar; expand steps such as **Start MCP Servers**	Did setup steps and MCP servers start? Which MCP tools were listed?
Firewall warning	Added to the PR body (new PR) or a comment (existing PR)	Which address was blocked and which command tried to reach it?
Plans	Plan Markdown produced by a planning custom agent; in VS Code the Plan agent persists its plan to `plan.md` in session memory (view with **Chat: Show Memory Files**)	Was the failure already present in the plan (wrong approach) or introduced during execution (drift)?
VS Code Agent Debug Logs PREVIEW	Chat view **...** > **Show Agent Debug Logs**, or **Developer: Open Agent Debug Logs**; Logs, Summary and Agent flow chart views	Chronological model requests, tool calls, customization discovery and errors; token usage and duration; which agent or subagent delegated an action

Evidence	Where (exact UI or command)	Answers the question
VS Code Chat Debug view	Chat view overflow menu > **Show Chat Debug View**, or **Developer: Show Chat Debug View**	Exactly which system prompt, user prompt, context, tool definitions and tool responses reached the model
Agent outputs	PR diff, PR description, review comments, generated files	Did the result meet acceptance criteria and constraints?
Workflow logs and artifacts	Actions tab > workflow > run summary > **Artifacts**; `gh run download RUN_ID -n NAME`; re-run with debug logging, or set `ACTIONS_STEP_DEBUG` / `ACTIONS_RUNNER_DEBUG` to `true`	What happened in CI or in a CLI-driven agent job, and what intermediate files did it produce?

In VS Code, capture is not retroactive: enable the experimental `chat.agentHost.agentDebugLog.enabled` setting (Agent Host sessions) or `github.copilot.chat.agentDebugLog.fileLogging.enabled` (Local harness sessions) before reproducing. You can then ask `/troubleshoot` a focused question about captured events, or export the session as OpenTelemetry (OTLP) JSON. The Chat Debug view sections map to root-cause questions: **System prompt** (did instructions and the agent definition arrive?), **Context** (was the needed file present?), **Response** (was the output complete?) and **Tool responses** (did each tool get the expected input and return a usable result?).

COMMON TRAP: WHAT A LOG LINE PROVES

Read evidence literally. A line showing that an MCP server configuration was *passed* to a session does not prove the server connected or that any tool was called. A successful model request tagged as an agent request does not prove which tools were authorised or that files were edited. A trimmed excerpt cannot tell you totals outside the lines shown. A "completed" session status does not prove that every command inside it succeeded. Exam statements that claim more than the excerpt shows are false, even if they are plausible.

Agent jobs that you run yourself in GitHub Actions (for example Copilot CLI in programmatic mode) should leave evidence that survives the run. Workflow logs and artifacts are kept for 90 days by default (configurable per repository), and re-running a non-deterministic agent may not reproduce the same behaviour, so the artifact from the *original* run is the evidence to analyse. Listing 4.5 captures a CLI transcript and logs and uploads them even when the agent step fails.

Listing 4.5: Running Copilot CLI in a workflow and preserving its transcript and logs as an artifact

```
name: agent-docs-refresh
on:
  workflow_dispatch:

permissions:
  contents: read

jobs:
  refresh:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - uses: actions/setup-node@v7
      - name: Install Copilot CLI
        run: npm install -g @github/copilot

      - name: Run agent
        env:
          COPILOT_GITHUB_TOKEN: ${{ secrets.COPILOT_CLI_TOKEN }}
        run: |
          mkdir -p .agent/logs
          copilot -p "Update docs/api.md to match the handlers in src/api. Do not edit other files." \
            --allow-tool='write' \
            --deny-tool='shell(git push)' \
            --no-ask-user \
            --log-level=debug \
            --log-dir=.agent/logs \
```

```
            --share=.agent/session-transcript.md

      - name: Upload agent evidence
        if: always()
        uses: actions/upload-artifact@v7
        with:
          name: agent-evidence-${{ github.run_id }}-${{ github.run_attempt }}
          path: .agent/
          include-hidden-files: true
          if-no-files-found: error
          retention-days: 30
```

Key details: `if: always()` uploads evidence even after a failed agent step; the artifact name includes `github.run_id` and `github.run_attempt` so re-runs do not collide; `include-hidden-files: true` is needed because `.agent` is a hidden directory (the action excludes hidden files by default, and its documentation tells you to validate hidden file contents before enabling this); `if-no-files-found: error` turns "the agent produced no logs" into a visible failure instead of a warning. `--share=PATH` writes the session to a Markdown file after a programmatic session, and `--log-dir` moves the CLI logs from the default `~/.copilot/logs/`.

PRODUCTION NOTE

Transcripts and debug exports can contain prompts, source code and tool outputs: sanitise them, keep retention short and never include credentials (console masking does not sanitise uploaded files). Files written by hooks in a cloud agent session are discarded with the sandbox; send hook output through an `http` hook to an allowlisted host to retain it.

4.2.2 Classify root causes, including reasoning errors, tool misuse, and context or environment issues

Once you have the evidence, put the failure in one class. The classes below follow the outline wording and split "tool" problems into misuse by the model versus a tool that is itself broken, because the fixes differ.

Root cause class	Definition	Typical evidence	Examples	Primary fix lever
Reasoning error	The agent had the right context and working tools but drew a wrong conclusion or chose a wrong approach.	Session log shows correct files read and tools succeeding, but the plan or diff is wrong.	Fixes the symptom not the cause; changes a test's expected value; declares success without running tests.	Sharper acceptance criteria; reviewed plan before implementation; validation steps in instructions.
Tool misuse	The model chose an inappropriate tool or called a tool with wrong or dangerous arguments.	Tool calls with wrong paths, wrong flags or wrong targets; destructive shell commands; tool used outside the task scope.	Uses shell `sed` to rewrite a workflow; calls a write-capable MCP tool instead of a read-only one.	Restrict tools (`tools` in the agent profile, MCP tool allowlists, CLI deny rules); `preToolUse` hooks; clearer tool guidance in instructions.
Tool integration or runtime failure	The tool or MCP server itself failed, independent of how sensibly it was called.	A `TypeError` inside a tool invocation; server missing from **Start MCP Servers** or failing in **MCP: List Servers**.	MCP server missing a runtime (`uv`, `pipx`); wrong server command; extension bug.	Fix server configuration or setup steps; update the server or extension; report the defect.

Root cause class	Definition	Typical evidence	Examples	Primary fix lever
Context issue	The information the agent needed was missing, stale, conflicting or not loaded.	Chat Debug view **Context** lacks the file; instructions absent from **System prompt**; `applyTo` glob does not match; customization diagnostics show an error; stale memory.	Wrong `applyTo` glob; conflicting personal and repository instructions; vague issue body.	Fix discovery and scope of instructions; add explicit context; prune or correct memory; rewrite the task.
Environment issue	The execution environment prevented correct work.	Firewall warning; install or build failures in the session log; authentication errors; setup step non-zero exit; timeout.	Missing dependencies; blocked registry host; secret stored as an Actions secret instead of an Agents secret; token scope; ruleset blocks the agent; session time limit.	`copilot-setup-steps.yml`; firewall allowlist; Agents secrets and variables; permission and ruleset bypass changes; smaller tasks.

EXAM TIP

Classify on the *first* point of divergence. If the firewall blocked a dependency install and the agent then wrote a broken workaround, the root cause is environmental. A tool error alone does not prove misuse: reasonable arguments plus a failure inside the tool runtime is an integration failure.

The decision table below maps the symptoms you are most likely to meet to a root-cause class and a documented fix. Learn it as a lookup: most scenario questions in this domain are one row of this table.

Symptom	Look first at	Likely class	Fix
Session completed and a PR was opened, but the PR has no file changes	The agent session log (tool calls, results, the agent's stated reason)	Depends on the log: reasoning (decided no change was needed), environment (edits or build failed), context (could not find the code)	Fix the class the log reveals; tighten acceptance criteria so "no change" is not a valid outcome.
A warning in the PR body names a blocked address and command	The firewall warning	Environment (network)	Add the host or URL to the repository or organisation custom allowlist, or preinstall the dependency in `copilot-setup-steps.yml` (setup steps are not subject to the agent firewall).
Session log appears stuck, then the session stops	Session log timeline	Environment (time budget) or reasoning (loop)	The session times out (59-minute maximum); retry by unassigning and reassigning the issue, and split the task.
PR shows no CI results after Copilot pushes	The merge box	Not a failure: approval gate	Review the diff, especially `.github/workflows/`, then click **Approve and run workflows**.
Copilot's pushes repeatedly fail CI checks	Session log: did it run tests and linters?	Context (no build or test instructions) or environment (dependencies missing)	Document build, test and lint commands in `.github/copilot-instructions.md`; preinstall dependencies with setup steps.
Agent edits tests or workflow files to get green checks	PR diff and session reasoning	Reasoning (goal misalignment)	State the constraint; CODEOWNERS plus code owner review on protected paths; optional `preToolUse` hook; required checks from a pinned source.

Symptom	Look first at	Likely class	Fix
A setup step failed and the agent then struggled to build	Setup steps in the session log	Environment	If a setup step exits non-zero, remaining setup steps are skipped and the agent starts anyway; fix the step and validate the workflow from the Actions tab.
MCP tool never used by the cloud agent	**Start MCP Servers** step	Integration failure or environment (missing runtime or secret)	Fix server config; install runtimes in setup steps; store keys as Agents secrets prefixed `COPILOT_MCP_`.
MCP tool not invoked in VS Code	**MCP: List Servers**; tool definitions in Chat Debug view	Integration (tool absent) or tool selection (tool present but not chosen)	Troubleshoot the server if absent; if present, name the tool and action in the prompt or agent instructions.
GitHub MCP server returns nothing for another repository	Tool response in the session log	Environment (token scope)	The built-in GitHub MCP server uses a token scoped read-only to the current repository; configure a fine-grained PAT or GitHub App only if wider access is justified.
Instructions file ignored	VS Code chat customization **Diagnostics**; Chat Debug view System prompt; References list	Context	Fix file name and location, frontmatter errors, or the `applyTo` glob.
Copilot does not react to a PR comment	Commenter's permission; PR state	Environment (permissions)	Only users with write access can trigger Copilot; it does not respond on merged or closed PRs.
Copilot cannot create or update PRs in a repository with rulesets	Ruleset configuration	Environment (policy)	Adjust incompatible rules or add Copilot as a bypass actor for that ruleset.

> **NOTE**
> The firewall applies only to processes the agent starts through its Bash tool; it does not directly apply to MCP server processes or Copilot setup steps, and it is not compatible with self-hosted runners or Windows, where you must disable it and supply your own network controls.

4.3 Tune agent behavior based on evaluation results

Tuning closes the loop. Follow the verification discipline from VS Code's troubleshooting guidance: change **one** relevant condition, repeat a comparable request with the same prompt, model and tools, and inspect the same event again to confirm the effect.

4.3.1 Revise instructions, workflows, or constraints

Instructions. Copilot on GitHub supports three kinds of repository instructions: repository-wide (`.github/copilot-instructions.md`), path-specific (`NAME.instructions.md` files in or below `.github/instructions/` with an `applyTo` glob in frontmatter), and agent instructions (`AGENTS.md` files anywhere in the repository, where the nearest one in the directory tree takes precedence, or a single `CLAUDE.md` or `GEMINI.md` at the root). Matching path-specific and repository-wide files are both used. Personal instructions take priority over repository, then organisation, but all relevant sets are sent, so avoid conflicts.

Support differs by feature and surface, which is a frequent exam detail:

Feature (on GitHub.com)	Instruction types read
Copilot cloud agent	Repository-wide, path-specific, agent (`AGENTS.md` , `CLAUDE.md` , `GEMINI.md`), organisation
Copilot code review	Repository-wide, path-specific, agent (`AGENTS.md` , `CLAUDE.md` , `GEMINI.md` , `REVIEW.md`), organisation
Copilot Chat	Personal, repository-wide, organisation
Copilot code review in VS Code or Visual Studio	Repository-wide only

For code review, Copilot reads repository instructions, agent instructions and skills from the **head branch** of the PR, so you can test an instruction change in the same PR that introduces it. Custom instructions for code review are enabled by default and can be disabled in repository settings. GitHub's code review tutorial gives concrete tuning rules: keep files short and focused (about 1,000 lines maximum per file), use headings and short imperative bullets, include correct and incorrect code examples, start with 10 to 20 specific instructions, then add instructions one at a time and test each with a real PR. Code review does not support instructions that change comment formatting or the overview comment, block merges, follow external links, or ask vaguely to "be more accurate".

Listing 4.6: Repository-wide instructions tuned after evaluation (.github/copilot-instructions.md)

```
# Repository standards for Copilot

## Build and validate (always run before finishing)
- Install: `npm ci`
- Unit tests: `npm test`
- Lint: `npm run lint` (must report zero errors)
- Trust these commands; search the repo only if they fail.

## Non-negotiable constraints
- Never edit files under `.github/workflows/` to make a check pass.
- Never delete, skip (`.skip`, `xit`) or weaken existing tests or assertions.
- If a test fails, fix the production code or explain in the PR why the test is wrong.

## Security review focus
- Flag hardcoded secrets, tokens or connection strings.
- Require parameterised queries; flag string-built SQL.
- Validate and sanitise all request input in `src/api/**`.

## Code quality
- Functions under 50 lines; descriptive names.
- New behaviour requires unit tests covering error paths.
```

Listing 4.7: Path-specific instructions with applyTo, used only by Copilot code review (.github/instructions/api-review.instructions.md)

```
---
applyTo: "src/api/**/*.ts,src/api/**/*.tsx"
excludeAgent: "cloud-agent"
---

# API review rules

- Every handler must return typed errors through `ApiError`; flag bare `throw new Error`.
- Flag any handler that reads `req.body` without schema validation.
- Flag new endpoints that are missing an entry in `docs/api.md`.

```ts
// Avoid
throw new Error("not found");

// Prefer
```
```

```
throw new ApiError(404, "order_not_found");
```
```

The optional `excludeAgent` keyword accepts `"code-review"` or `"cloud-agent"` and stops the named feature from reading the file; if it is omitted, both features use it. Remember the glob rules: `*.py` matches only the current directory, `**/*.py` matches recursively, and multiple patterns are comma-separated inside one string.

**Workflows.** If evaluation shows the agent spending its time on trial-and-error installs, or failing because a private dependency cannot be fetched, move that work into `.github/workflows/copilot-setup-steps.yml`. The file must contain a single job named `copilot-setup-steps` (otherwise it is not picked up), must be on the default branch to take effect, and only `steps`, `permissions`, `runs-on`, `services`, `snapshot` and `timeout-minutes` can be customised. Copilot receives its own token, so keep `permissions` minimal.

**Listing 4.8: Setup steps that remove the most common environment root causes**

```yaml
name: "Copilot Setup Steps"
on:
  workflow_dispatch:
  push:
    paths: [.github/workflows/copilot-setup-steps.yml]
  pull_request:
    paths: [.github/workflows/copilot-setup-steps.yml]

jobs:
  copilot-setup-steps:            # job name is mandatory
    runs-on: ubuntu-latest
    timeout-minutes: 30           # shorter than the 59-minute ceiling
    permissions:
      contents: read
    steps:
      - uses: actions/checkout@v6
      - uses: actions/setup-node@v7
        with:
          node-version: "22"
          cache: "npm"
      # NPM_TOKEN is an Agents secret: it is exposed to setup steps
      # as an environment variable and read by the committed .npmrc
      - name: Install dependencies (incl. private registry)
        run: npm ci
```

Secrets used by the agent environment are configured as **Agents** secrets and variables (repository **Settings > Secrets and variables > Agents**, or at organisation level). Data you do not want Copilot to access should stay in Actions secrets, which Copilot cannot read in its sessions or setup steps. Agents secrets and variables are exposed as environment variables, except those prefixed `COPILOT_MCP_`, which are available only to MCP servers. Validate a setup-steps change by running the workflow manually from the Actions tab.

Constraints. Instructions are persuasive; constraints are enforced. When evaluation shows an agent repeatedly crossing a line, add a control that does not depend on the model complying: the ruleset and CODEOWNERS pattern from Listings 4.2 and 4.3, push restrictions on protected paths, and, for defence in depth inside the session, a hook. Copilot cloud agent loads hooks from `.github/hooks/*.json` in the cloned repository and honours `bash` (or `command`) entries. A `preToolUse` hook can return `permissionDecision` of `"allow"` or `"deny"` (a reason is required when denying); under cloud agent, `"ask"` is treated as `"deny"` because no user is present. Command `preToolUse` hooks fail closed on a crash or non-zero exit, but a timeout is fail-open.

Listing 4.9: A preToolUse hook that denies file edits under .github/workflows/ (.github/hooks/protect-ci.json and script)

```json
{
  "version": 1,
  "hooks": {
    "preToolUse": [
      {
        "type": "command",
```
```

```
        "matcher": "edit|create",
        "bash": "./scripts/hooks/protect-ci.sh",
        "timeoutSec": 10
      }
    ]
  }
}

# scripts/hooks/protect-ci.sh  (must be executable, with a shebang)
#!/usr/bin/env bash
INPUT=$(cat)
if printf '%s' "$INPUT" | grep -q '\.github/workflows/'; then
  echo '{"permissionDecision":"deny","permissionDecisionReason":"CI workflows are protected. Fix the code or
tests, or explain the needed CI change in the PR description."}'
else
  echo '{}'
fi
```

COMMON TRAP

A hook that matches only `edit|create` does not see shell commands, and a hook that times out lets the call proceed. Hooks reduce wasted iterations and give the agent immediate feedback, but the authoritative boundary is still outside the agent: code owner review, required checks and push rules. Never weaken those because a hook exists.

4.3.2 Refine memory usage

Memory makes agents better over time, and it is also a source of stale or wrong context. Tuning memory means deciding what should be remembered, where, for how long, and when a remembered fact should graduate into reviewed, version-controlled instructions.

Copilot Memory PREVIEW stores repository-level facts (coding conventions, architectural decisions, build commands) and user-level preferences. It is used by Copilot cloud agent, Copilot code review, Copilot CLI and agentic autofix, and facts learned by one feature can be used by another. Important details for evaluation and tuning:

- Repository facts are stored with **citations** to supporting code and are **validated against the current branch** before use; only validated facts are used.
- Repository facts come only from users with write access, stay in that repository, and can be reviewed and deleted by repository owners.
- Copilot code review uses repository-level facts only, never user-level preferences. Copilot CLI applies repository facts and the initiating user's preferences.
- Unused facts and preferences are deleted automatically after **28 days**; successful validation and use may reset the timer.
- Memory is enabled per user, not per repository; organisation and enterprise plans need an administrator policy first.

VS Code memory is local. The built-in memory tool stores notes in three scopes: user (`/memories/`, persists across workspaces, first 200 lines loaded into every session), repository (`/memories/repo/`, workspace-scoped) and session (`/memories/session/`, current conversation only, used by the Plan agent for `plan.md`). Use **Chat: Show Memory Files** to inspect and **Chat: Clear All Memory Files** to reset. In Copilot CLI, the `memory` permission kind can be allowed or denied like any other tool.

Evaluation finding	Memory action
The same correction is given in every session ("we use pnpm, not npm")	Make it durable: a repository fact in Copilot Memory, or better, a line in `.github/copilot-instructions.md` so it is reviewed and shared.
A useful local repository memory has become team knowledge	Verify it against the code, move it into project docs or instructions through normal review, then remove the local copy to avoid conflicting versions.

Evaluation finding	Memory action
Agent applies an outdated convention	Delete the stale fact (repository owners can delete Copilot Memory facts; ask the VS Code agent to update or delete a memory file), and correct the instructions.
Code review ignores a personal preference	Expected: code review uses repository facts and instructions only. Put team review rules in instruction files.

4.3.3 Refine tool usage and tool access

Tool access is both a quality lever (fewer, better-described tools make selection easier) and a safety lever (least privilege). Custom agents are the main mechanism. An agent profile is a Markdown file with YAML frontmatter, stored in `.github/agents/` for a repository (VS Code detects `.md` files there; the `.agent.md` extension is the convention). In the profile, `description` is required, and the `tools` property filters the tools available:

- Omit `tools` or use `["*"]` to enable all available tools, including MCP tools configured in the profile or repository settings.
- List specific tools or aliases to enable only those, for example `["read", "search", "edit"]`.
- Use `tools: []` to disable all tools.
- Reference MCP tools as `server-name/tool-name`, or all tools of a server as `server-name/*`. Unrecognised tool names are ignored, so a misspelled tool silently disappears; check the tool definitions in the Chat Debug view when tuning.

The documented aliases are `execute` (shell), `read`, `edit`, `search`, `agent` (invoke another custom agent), `web` and `todo`; `web` and `todo` are currently not applicable for cloud agent. The out-of-the-box MCP servers for cloud agent are `github` (read-only tools, token scoped to the source repository) and `playwright` (localhost only). For repository MCP configuration, GitHub strongly recommends allowlisting specific read-only tools, because the agent uses MCP tools autonomously without asking for approval.

Listing 4.10: Implementation planner agent profile tuned after evaluation (.github/agents/implementation-planner.agent.md)

```
---
name: implementation-planner
description: Creates implementation plans with success criteria and constraints; never changes source code
tools: ["read", "search", "github/*"]
disable-model-invocation: true
---

You are a technical planning specialist. Return the plan in your response; you cannot edit files.

For every plan include:
- Overview, with explicit success criteria ("what does done look like?")
- Technical approach and trade-offs
- Phased tasks with Small/Medium/Large estimates and dependencies
- Considerations: assumptions, constraints, risks
- Not included (out of scope)

Rules:
- Read the linked issue and related code before planning.
- If requirements are ambiguous, list open questions instead of guessing.
```

The customization library version uses `tools: ["read", "search", "edit"]`. If evaluation shows planner sessions editing source files, a "do not edit code" sentence is only guidance; removing `edit` is enforcement. The tuned profile keeps `read` and `search`, adds the built-in read-only `github` MCP server for issue context, omits `execute`, and sets `disable-model-invocation: true` so cloud agent never selects it automatically. The trade-off is that the plan is returned rather than committed.

> **NOTE**
> In VS Code, a `handoffs` entry with `send: false` (the default) only stops the prompt being auto-submitted; it does not prevent switching agents, so it does not by itself separate planning from implementation. The `agents` property limits which subagents can be invoked. Cloud agent ignores `handoffs` and `argument-hint`, and uses the profile version (Git commit SHA) the PR started with.

For agents you run with Copilot CLI, tune access on the command line. `--allow-tool` and `--deny-tool` take permission patterns of the form `Kind(argument)` with kinds `shell`, `write`, `read`, `url`, `memory` or an MCP server name. Deny rules always take precedence over allow rules, even when `--allow-all` is set. `--available-tools` and `--excluded-tools` control which tools the model can see at all.

Listing 4.11: Copilot CLI permissions tuned after an evaluation found unwanted pushes and workflow edits

```
# Before: everything auto-approved
copilot -p "Fix the failing unit tests in src/billing" --allow-all-tools

# After: narrow, explicit, deny-first
copilot -p "Fix the failing unit tests in src/billing. Do not change tests." \
  --allow-tool='shell(npm:*)' \
  --allow-tool='shell(git:*)' \
  --deny-tool='shell(git push)' \
  --allow-tool='write' \
  --deny-tool='write(.github/workflows/ci.yml)' \
  --deny-tool='memory' \
  --no-ask-user
```

`--deny-tool='write(PATH)'` matches an exact path or trailing segment without glob support, so directory-wide protection still belongs to rulesets and code owner review.

> **EXAM TIP**
> When a scenario says an agent "has more access than it needs" or "used a tool it should not", the answer is almost always to *remove or narrow* a tool (profile `tools`, MCP `tools` allowlist, CLI deny rule), not to add a sentence to the prompt. When it says the agent "does not know how to" do something, the answer is instructions or setup steps.

4.4 Key terms

Task contract	The written definition of an agent task: outcome, acceptance criteria, scope, operational constraints and environment limits. The basis for every later evaluation.
Evaluation signal	Any observable evidence used to judge agent work. Quantitative signals are counts, rates, durations and pass/fail states; qualitative signals are review comments, session-log reasoning and feedback.
Require code scanning results	A ruleset rule that blocks merges when a required tool (for example CodeQL) finds alerts at or above configured thresholds, is still analysing, or is not configured.
Built-in validation tools	CodeQL, GitHub Advisory Database checks, secret scanning and Copilot code review run by Copilot cloud agent on its own output before finishing a PR; toggled under Settings > Copilot > Cloud agent.
Session log	The record of a Copilot cloud agent session showing internal reasoning, tool usage, validation, token usage and session length; linked from the PR timeline and from each agent commit.
Root cause class	The category of the first point of divergence: reasoning error, tool misuse, tool integration or runtime failure, context issue, or environment issue.
`excludeAgent`	Frontmatter keyword in a path-specific instructions file that stops either `"code-review"` or `"cloud-agent"` from using the file.
Agents secrets and variables	Dedicated secret and variable type for Copilot cloud agent at organisation or repository level; names

	prefixed `COPILOT_MCP_` are available only to MCP servers.
Copilot Memory	Preview feature storing cited, validated repository facts and user preferences, used by cloud agent, code review (repository facts only), CLI and agentic autofix; unused entries expire after 28 days.
`preToolUse` hook	Hook event that runs before each tool call and can allow, deny or modify it; command variants fail closed on errors but are fail-open on timeout.

4.5 Additional practice questions

Practice 4.1 MULTIPLE CHOICE

Your repository uses Copilot code review on GitHub.com and Copilot cloud agent. You want detailed React accessibility rules to be applied when Copilot code review reviews files under `src/components/`, but you do not want Copilot cloud agent to read those rules when it implements features. You need to configure this with the least effort. What should you do?

A. Add the rules to `.github/copilot-instructions.md` under a "Code review" heading.
B. Create `.github/instructions/a11y.instructions.md` with `applyTo: "src/components/**"` and `excludeAgent: "cloud-agent"`.
C. Create `.github/agents/a11y-reviewer.agent.md` and select it when requesting a review.
D. Create `.github/instructions/a11y.instructions.md` with `applyTo: "src/components/**"` and `excludeAgent: "code-review"`.

Practice 4.2 MULTIPLE RESPONSE

Copilot cloud agent opens a pull request whose body contains a warning that a request to `npm.internal.contoso.com` made by `npm ci` was blocked. The agent then failed to run the tests. You need the agent to be able to install the internal packages on future tasks. Each correct answer presents a complete solution. Which TWO actions should you take?

A. Add `npm.internal.contoso.com` to the repository's custom firewall allowlist on the "Internet access" settings page.
B. Add an instruction to `.github/copilot-instructions.md` telling Copilot to retry `npm ci` three times.
C. Install the dependencies in a `copilot-setup-steps` job, with the registry token stored as an Agents secret.
D. Enable step debug logging by setting `ACTIONS_STEP_DEBUG` to `true`.
E. Remove `search` from the tools of the custom agent that was used.

Practice 4.3 MULTIPLE CHOICE

A Copilot cloud agent session log shows the agent calling a tool of the built-in GitHub MCP server with well-formed arguments to read issues from a different repository in your organisation. Each call returns no data, and the agent eventually writes code based on assumptions. You need to classify the root cause for your incident report. Which class is most accurate?

A. Reasoning error, because the agent wrote code based on assumptions.
B. Tool misuse, because the agent called the wrong MCP tool.
C. Environment issue (access scope), because the built-in GitHub MCP server's token is scoped read-only to the current repository.
D. Tool integration failure, because the GitHub MCP server crashed.

Practice 4.4 MULTIPLE CHOICE

Engineering leadership asks for a monthly comparison of median time to merge for pull requests created by Copilot versus all pull requests, across every organisation in the enterprise. You need a documented data source that provides this. What should you use?

A. The Copilot usage metrics API or dashboards, using the pull request lifecycle metrics.
B. The REST API endpoints for Copilot user management, using `last_activity_at`.
C. The GitHub Actions usage metrics for each repository.
D. Exported VS Code Agent Debug Logs from each developer.

Practice 4.5 MULTIPLE CHOICE

A developer with Copilot Memory enabled has repeatedly told Copilot that they prefer early returns over nested conditionals. Copilot CLI now follows this preference, but Copilot code review never comments on nested conditionals in the developer's PRs. You need the review behaviour to apply to the whole team. What should you do?

A. Ask the developer to repeat the preference in a PR comment that mentions `@copilot`.
B. Add the rule to the repository's custom instructions, because Copilot code review uses repository-level facts and instructions, not user-level preferences.
C. Ask an organisation owner to enable Copilot Memory for the repository.
D. Wait 28 days for the preference to be promoted to a repository fact.

Practice 4.6 MULTIPLE CHOICE

A Copilot cloud agent task to migrate 140 API handlers to a new framework consistently stops before finishing. The session log shows steady progress until the session ends. You need the migration to complete reliably. What should you do?

A. Set `timeout-minutes: 120` in the `copilot-setup-steps` job.
B. Disable the built-in validation tools so the agent finishes faster.
C. Split the migration into smaller, focused issues, each with its own acceptance criteria.
D. Upgrade the agent to a larger runner so the session limit increases.

Practice 4.7 MULTIPLE CHOICE

You add a third-party MCP server to your repository's MCP configuration for Copilot cloud agent. Its API key is stored as a GitHub Actions repository secret named `VENDOR_KEY`. In the session log, the **Start MCP Servers** step reports an authentication failure. You need to fix the configuration. What should you do?

A. Move the key to an Agents secret named `COPILOT_MCP_VENDOR_KEY` and reference it from the MCP configuration.
B. Rename the Actions secret to `COPILOT_VENDOR_KEY`.
C. Add the vendor's host to the firewall allowlist.
D. Add `permissions: secrets: read` to `copilot-setup-steps.yml`.

Practice 4.8 MULTIPLE CHOICE

Evaluation shows that a `docs-writer` custom agent sometimes runs shell commands and calls MCP tools that create issues. The agent only needs to read code and edit Markdown files. You need to apply least privilege in the agent profile. Which frontmatter should you use?

A. Omit the `tools` property.
B. `tools: ["*"]`
C. `tools: ["read", "search", "edit"]`
D. `tools: []`

Practice 4.9 MULTIPLE CHOICE

Agent-authored PRs in a repository with CodeQL default setup have been merged even though code scanning reported new high-severity security alerts. You need merges to be blocked when CodeQL finds new alerts of high or critical security severity, without blocking on lower severities. What should you configure?

A. A branch ruleset with "Require code scanning results", tool CodeQL, and security alerts threshold "High or higher".
B. A required status check named `CodeQL` with "any source".
C. Dependabot security updates for the repository.
D. A path-specific instructions file telling Copilot to fix all CodeQL alerts.

Practice 4.10 MULTIPLE CHOICE

A nightly workflow runs Copilot CLI with `copilot -p` to refactor code. When the agent step fails, the team has no record of what the agent did, and re-running produces different behaviour. You need to retain the transcript of every run, including failed ones, for post-hoc analysis. What should you add?

A. `--log-level=all` only, and re-run failed jobs with debug logging enabled.
B. `--share=transcript.md` on the CLI call, and an `actions/upload-artifact` step with `if: always()` that uploads the transcript.
C. An `actions/download-artifact` step after the agent step.
D. A `sessionEnd` command hook that writes the transcript to `~/transcript.md`.

Answers and explanations

4.1: B. Path-specific instructions are read by both Copilot code review and Copilot cloud agent on GitHub.com unless `excludeAgent` names one of them. `excludeAgent: "cloud-agent"` keeps the rules for code review only. D does the opposite; A applies everywhere; agent profiles do not configure code review.

4.2: A and C. Both are documented fixes for a firewall block: allowlist the host (domain rules also allow subdomains), or install dependencies in setup steps, which run before the agent and are not subject to the agent's firewall. Private registry credentials belong in Agents secrets. B, D and E do not change network access.

4.3: C. Classify on the first point of divergence. The tool call was reasonable and the server worked, but its token is scoped to the current repository, so data from other repositories is unavailable by design. The assumption-based code is a consequence. A PAT or GitHub App with wider read access is the fix, if justified.

4.4: A. Copilot usage metrics include pull request lifecycle metrics (creation and merge counts, median time to merge, review suggestion activity) and let you compare Copilot-created PRs with all PRs at organisation or enterprise level. The user management API is the source of truth for seats, not PR outcomes.

4.5: B. Copilot code review applies repository-level facts only; user-level preferences are not used during review. A team-wide review rule belongs in reviewed repository instructions. Memory is enabled per user, not per repository, and the 28-day period is an expiry for unused entries, not a promotion mechanism.

4.6: C. Each session has a maximum execution time of 59 minutes that cannot be extended or bypassed; `timeout-minutes` can only shorten it (maximum 59). Larger runners do not change the limit.

4.7: A. MCP servers for cloud agent receive only Agents secrets and variables whose names start with `COPILOT_MCP_`. Actions secrets are not available to the agent's sessions or setup steps. The error is authentication, not a blocked host, and `secrets` is not a `permissions` scope.

4.8: C. A specific list enables only those tools: `read`, `search` and `edit` without `execute` or any MCP tools. Omitting `tools` or using `["*"]` enables everything, and `[]` disables all tools, so the agent could not read or edit.

4.9: A. "Require code scanning results" evaluates the tool's results against configurable alert and security-alert thresholds and also blocks while analysis is in progress or not configured. A plain status check only reports job success; B to D do not gate on severity.

4.10: B. `--share=PATH` writes the session to Markdown after a programmatic session, and `if: always()` uploads it even when the agent step fails. Re-running does not recover the original behaviour, and `download-artifact` retrieves rather than uploads.

Key takeaways

- Write agent tasks as contracts: problem, observable outcome, acceptance criteria, file scope, explicit constraints (no workflow edits, no weakened tests) and out-of-scope items. Plans must state success criteria before implementation starts.
- Fit tasks to platform limits: 59-minute sessions, one repository, one branch and one PR per task, firewall-restricted network, and human approval before workflows run and before merge.
- Combine quantitative signals (check status, test results, alerts, iteration count, merge rate, time to merge, tokens, session length) with qualitative ones (review comments, session reasoning). Copilot usage metrics provide PR lifecycle metrics at organisation and enterprise scope.
- Align metrics with intent by enforcing outside the agent: CODEOWNERS plus code owner review for workflows, tests and agent configuration; required status checks with pinned sources; "Require code scanning results" thresholds.
- Cloud agent runs CodeQL, Advisory Database dependency checks, secret scanning and Copilot code review on its own work by default, without a GHAS licence; gate merges on independent PR checks as well.
- Evidence first: session logs (and **Start MCP Servers**), firewall warnings, plans, VS Code Agent Debug Logs and Chat Debug view, Actions logs and artifacts from the original run. Read logs literally; do not infer more than they show.
- Classify on the first divergence: reasoning error, tool misuse, tool integration failure, context issue or environment issue. Each class has a different fix.
- Tune one lever at a time and re-test: instructions (repository-wide, path-specific with `applyTo` and `excludeAgent`, AGENTS.md), setup steps and Agents secrets, rulesets and hooks, memory hygiene, and least-privilege tool lists.

Official documentation references

- Study guide for Exam GH-600: Developing in Agentic AI Systems (Microsoft Learn)
- Implementation planner (customization library)
- Custom agents configuration
- Best practices for using GitHub Copilot to work on tasks
- Troubleshooting GitHub Copilot cloud agent
- Managing agent sessions
- GitHub Copilot on GitHub.com (limitations and compatibility)
- Risks and mitigations for GitHub Copilot cloud agent
- Configuring settings for GitHub Copilot cloud agent
- Review output from Copilot
- Building guardrails for GitHub Copilot cloud agent

- Customizing or disabling the firewall for GitHub Copilot
- Configure the development environment (copilot-setup-steps.yml)
- Configure secrets and variables for Copilot cloud agent
- Configure MCP servers for your repository
- Support for different types of custom instructions
- Adding repository custom instructions for GitHub Copilot
- Using custom instructions to unlock the power of Copilot code review
- GitHub Copilot code review
- About GitHub Copilot Memory
- GitHub Copilot usage metrics
- GitHub Copilot hooks reference
- Copilot customization cheat sheet
- GitHub Copilot CLI command reference
- Automating Copilot CLI with GitHub Actions
- Available rules for rulesets
- REST API endpoints for rules
- Set code scanning merge protection
- About code owners
- Configuring default setup for code scanning
- CodeQL code scanning for compiled languages
- Configuring the dependency review action
- Dependency review
- Dependabot alerts
- Secret scanning
- Push protection
- Downloading workflow artifacts
- Enabling debug logging
- actions/upload-artifact README
- Debug chat interactions (VS Code)
- Troubleshoot AI in Visual Studio Code
- Custom agents in VS Code
- Use memory with agents in VS Code

DOMAIN 5

Orchestrate multi-agent coordination

Exam weight: 15–20%

This domain tests whether you can run several agents together safely: choosing how they coordinate, keeping their work apart while they run in parallel, seeing what each one did, recovering when one of them fails, and changing the line-up of agents over time without losing history. The exam expects you to know the concrete GitHub controls that make this possible, not just the theory. That means the `agents`, `handoffs`, `user-invocable` and `disable-model-invocation` properties of custom agent profiles, subagents in VS Code and Copilot CLI (including `/fleet`), GitHub Actions `needs`, `strategy.matrix` and `concurrency`, Git branches and worktrees, session logs, the `gh agent-task` and `gh run` commands, and Git-based versioning of agent profiles.

Study approach: build a three-agent workflow (planner, implementer, reviewer) in a scratch repository. Run it once with handoffs, once with a coordinator that calls subagents, and once as a GitHub Actions fan-out/fan-in. Then break things on purpose: make two agents edit the same file, let one stall, and revert an agent profile. Most exam scenarios in this domain are variations of those exercises.

Skill group	Outline bullet	Section
Operate and manage multi-agent workflows	Apply an orchestration pattern to coordinate multiple agents	5.1.1
	Configure agent isolation for parallel execution	5.1.2
	Detect and resolve agent conflicts, including overlapping code changes, duplicated effort, and contradictory outputs	5.1.3
Configure observability for multi-agent behavior by using logs, artifacts, and operational signals	Configure multi-agent workflows to produce artifacts suitable for review and audit	5.2.1
	Document key decisions, handoffs, and outcomes across agents	5.2.2
	Perform post-hoc analysis of multi-agent behavior	5.2.3
Detect and respond to multi-agent failures and degraded behavior	Identify failed, partial, or stalled agent executions	5.3.1
	Respond to degraded behavior or coordination across agents	5.3.2
	Implement multi-agent recovery patterns, including rollback and human-in-the-loop	5.3.3
Manage the lifecycle of agents within multi-agent workflows	Add agents to existing multi-agent workflows	5.4.1
	Update, reconfigure, or replace agents without disrupting active workflows	5.4.2
	Retire agents while preserving auditability and workflow continuity	5.4.3

5.1 Operate and manage multi-agent workflows

A multi-agent workflow splits a goal into units of work and gives each unit to an agent that has the context, tools and model suited to it. GitHub Docs describe the typical coordinated workflow as five steps: break the goal into smaller tasks, identify dependencies between them, assign each task to a subagent with the context and tools it needs, run independent tasks in parallel, then review and combine the results. The orchestrator is usually the main agent that delegates to subagents. Everything in this section is a way of implementing one or more of those five steps with GitHub tooling.

Before you add agents, check that you need them. The Azure Architecture Center guidance on agent orchestration patterns says to start with the right level of complexity: if a single agent with enough tools can solve the task reliably, coordination overhead often exceeds the benefit of splitting it up. GitHub Docs make the same point about `/fleet`: every subagent talks to a model independently, so delegation can use more GitHub AI Credits, and if the work is inherently sequential, parallelism gives no benefit.

5.1.1 Apply an orchestration pattern to coordinate multiple agents

The Microsoft Learn architecture guide names five fundamental patterns: sequential, concurrent, group chat, handoff and magentic. The GH-600 outline focuses on the patterns you can build directly with GitHub Copilot features. The table below maps each pattern to the GitHub mechanism that implements it.

Pattern	How it coordinates	Use it when	GitHub implementation	Watch out for
Sequential handoff (pipeline, prompt chaining)	Stages run in a fixed order. Each stage takes the previous stage's output as input.	Each stage depends on the last one, and you want a human checkpoint between stages (plan, then implement, then review).	VS Code `handoffs` in `.agent.md`; Actions jobs chained with `needs`; separate cloud agent tasks run in sequence.	Failures in early stages propagate. No parallelism.
Orchestrator with subagents (coordinator and worker)	One coordinator keeps the goal, delegates focused tasks to workers in their own context, then combines what they return.	A task needs research, implementation and review by different specialists, but one agent should own the result.	Coordinator `.agent.md` with the `agent` tool and an `agents` list; Copilot CLI subagents; Copilot SDK `customAgents`.	Workers do not inherit conversation history. Coordinator context grows with every result.
Fan-out/fan-in (concurrent, scatter-gather)	Several agents work on the same input at the same time. An aggregator merges, compares or votes on their results.	You want independent perspectives (for example correctness and test coverage reviews) or independent producers whose outputs are merged.	Actions jobs with no dependencies between them, then a merge job with `needs: [a, b]`; "use two subagents to review" prompts; `/fleet`.	Contradictory results need a resolution rule. Resource use spikes.
Parallel per-item (map, map-reduce)	The same agent definition runs once per item (file, package, service, issue).	The work is a list of independent items with the same procedure.	`strategy.matrix` in Actions; `/fleet`; a recursive agent that lists itself in `agents`; one cloud agent task per issue.	Items must truly be independent. Matrix limit of 256 jobs per workflow run.

EXAM TIP

Match the wording of the scenario to the pattern. "Each step reviews the output of the last one before continuing" is a sequential handoff. "One agent decides which specialists to call and assembles the answer" is an orchestrator with subagents. "Run A and B at the same time, then merge" is fan-out/fan-in. "Do the same thing for each of these 40 packages" is parallel per-item.

Orchestrator with subagents in VS Code

In VS Code, a subagent is a separate agent invocation that works in its own context and returns a focused result to the main agent. In a Local harness session the main agent launches it with the `runSubagent` tool, which is part of the `agent` tool set. A coordinator custom agent needs two things in its front matter:

- `tools` must include `agent`. The VS Code docs say: if you specify `agents`, ensure the `agent` tool is included in the `tools` property. Without it the coordinator cannot start a subagent at all.
- `agents` lists which custom agents may run as subagents. `agents: ['Codebase Researcher', 'Reviewer']` permits only those agents; `agents: ['*']` or omitting the property permits all available agents; `agents: []` prevents subagent use.

Two more properties control how each worker can be reached:

- `user-invocable` controls whether the agent appears in the agents dropdown. Default `true`. Set it to `false` for a helper that should only run as a subagent. It does not affect subagent invocation.
- `disable-model-invocation` controls whether other agents can invoke this agent as a subagent. Default `false`. However, explicitly listing an agent in a coordinator's `agents` property overrides that agent's `disable-model-invocation: true`.

The older `infer` property is deprecated in VS Code and replaced by these two properties. Agent names are case-sensitive: use the exact `name` from the agent definition (or the file name if no `name` is set).

Listing 5.1: A coordinator that may call only two named workers (VS Code, `.github/agents/`)

```
# .github/agents/feature-coordinator.agent.md
---
name: Feature Coordinator
description: Implement a feature with delegated research and review
tools: ['agent', 'edit', 'read', 'search']
agents: ['Codebase Researcher', 'Reviewer']
---
For each feature request:
1. Ask Codebase Researcher for relevant files, patterns and open questions.
   Pass the goal, constraints and the decisions already made.
2. Implement the change yourself, using the research findings.
3. Ask Reviewer to check the changed files. Pass the requirements and
   a summary of your changes, because Reviewer has no chat history.
4. Address the findings, then summarise the change and remaining risks.

# .github/agents/codebase-researcher.agent.md
---
name: Codebase Researcher
description: Find relevant code and explain existing patterns
user-invocable: false
tools: ['read', 'search']
---
Research the requested topic without changing files.
Return file paths, existing patterns and unanswered questions.

# .github/agents/reviewer.agent.md
---
name: Reviewer
description: Review changes for correctness and missing tests
user-invocable: false
tools: ['read', 'search']
---
Review the supplied files and change summary without editing files.
Report correctness issues and missing test coverage with file references.
```

Note the division of privileges: the workers have read-only tools and the coordinator holds the edit tools. This is the coordinator and worker example from the VS Code subagents documentation, and it is a good default because only one agent can change files.

Points about Local subagents that the exam likes to test:

- **No inherited history.** A Local subagent does not inherit the main conversation history, so the delegated task must be self-contained: goal, context, allowed actions and expected result.
- **Stateless invocations.** The main agent cannot send follow-up messages to the same subagent. Further work needs a new invocation with the relevant context. Local subagents also cannot use the built-in tools for clarifying questions or todo lists.
- **Instructions and tools.** A subagent inherits the main agent's instructions and selected tools unless you run a custom agent, which supplies its own instructions and can override tools and model.
- **Model selection order.** An explicit model passed to `runSubagent`, then the custom agent's `model`, then Auto (if `chat.subagents.defaultToAuto` is on and conditions apply), then the main conversation's model. A selection above

the main model's cost tier does not run.

- **Nesting.** Subagents cannot start further subagents unless you enable `chat.subagents.allowInvocationsFromSubagents` (off by default). Nesting is limited to a depth of five. A recursive agent lists itself in its own `agents` property and must have a stopping condition.

Sequential handoffs in VS Code

Handoffs connect agents into a guided sequence. After a response completes, VS Code shows handoff buttons; selecting one switches to the target agent with the conversation context and a pre-filled prompt. Each entry under `handoffs` has `label` (button text), `agent` (target agent identifier), `prompt` (text to send), optional `send` (auto-submit, default `false`) and optional `model` (qualified name in the form `Model Name (vendor)`, for example `GPT-5 (copilot)`).

Listing 5.2: Planner, implementer and reviewer linked with handoffs

```
# .github/agents/planner.agent.md
---
name: Planner
description: Produce an implementation plan without editing code
tools: ['search', 'read', 'web/fetch']
handoffs:
  - label: Start Implementation
    agent: Implementer
    prompt: Implement the plan above. Keep to the files it lists.
    send: false
---
Write a plan with Overview, Requirements, Implementation Steps and Testing.
Do not edit files.

# .github/agents/implementer.agent.md
---
name: Implementer
description: Implement an approved plan with minimal, focused edits
tools: ['edit', 'read', 'search', 'execute']
handoffs:
  - label: Request Review
    agent: Reviewer
    prompt: Review the changes made in the previous step against the plan.
    send: true
---
Follow the approved plan. Record any deviation from it in your reply.

# .github/agents/reviewer.agent.md  (target of the second handoff)
---
name: Reviewer
description: Review changes for correctness and missing tests
tools: ['read', 'search']
handoffs:
  - label: Fix Findings
    agent: Implementer
    prompt: Address the review findings listed above.
---
```

The first handoff uses `send: false`, so the developer sees the plan, reads the pre-filled prompt and decides whether to proceed: a natural human checkpoint between planning and action. The second uses `send: true`, so moving from implementation to review happens with one click and no editing. The `agent` value must identify the target agent exactly; agent names are case-sensitive, so `implementer` does not match an agent named `Implementer`.

COMMON TRAP

Handoffs are a VS Code (IDE) feature. The GitHub Docs custom agents configuration reference states that `argument-hint` and `handoffs` are not supported for Copilot cloud agent on GitHub.com and are ignored. If a scenario needs planner, implementer and reviewer stages on GitHub.com, model them as separate cloud agent tasks, separate pull requests or Actions jobs, not as handoffs.

NOTE

Handoffs and subagents are different mechanisms. A handoff switches the active agent in the same chat after the user selects a button. A subagent is invoked by the main agent through the `agent` tool, runs in its own context, and returns a result. A coordinator does not need a handoff entry to call a subagent, and a handoff does not need the `agent` tool.

Fan-out/fan-in and per-item parallelism with GitHub Actions

Actions gives you a deterministic, auditable orchestrator. Jobs run in parallel by default. `jobs.<job_id>.needs` makes a job wait until the listed jobs complete successfully; if a needed job fails or is skipped, the dependent job is skipped too unless it uses a condition such as `always()`. `strategy.matrix` creates one job per combination of values, with a maximum of 256 jobs per workflow run. `fail-fast` (default `true`) cancels the rest of the matrix when one leg fails, and `max-parallel` caps how many legs run at once. Agents running in different jobs do not share a file system, so outputs move between them as artifacts (`actions/upload-artifact` and `actions/download-artifact`) or job outputs (`needs.<job_id>.outputs.<name>`).

Listing 5.3: Per-item fan-out with a matrix, a second independent producer, and a fan-in merge job

```
name: agent-review-fanout
on:
  workflow_dispatch:

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: false

jobs:
  package-review:                       # parallel per-item
    runs-on: ubuntu-latest
    timeout-minutes: 30
    strategy:
      fail-fast: false                  # one failing package must not cancel the others
      max-parallel: 4
      matrix:
        package: [api, web, worker, billing]
    steps:
      - uses: actions/checkout@v4
      - run: ./agents/review_package.sh "${{ matrix.package }}" > review-${{ matrix.package }}.md
      - uses: actions/upload-artifact@v4
        with:
          name: review-${{ matrix.package }}
          path: review-${{ matrix.package }}.md

  risk-scan:                            # independent producer, runs at the same time
    runs-on: ubuntu-latest
    timeout-minutes: 30
    steps:
      - uses: actions/checkout@v4
      - run: ./agents/risk_scan.sh > risk.md
      - uses: actions/upload-artifact@v4
        with:
          name: risk
          path: risk.md

  merge-plan:                           # fan-in
    needs: [package-review, risk-scan]
    if: ${{ always() }}                 # still aggregate partial results
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/download-artifact@v5
        with:
          path: inputs
```

```
      - run: ./agents/merge_plan.sh inputs > plan.md
      - run: cat plan.md >> "$GITHUB_STEP_SUMMARY"
```

Read the listing as a pattern, not as fixed syntax. The fan-out is the absence of `needs` between `package-review` and `risk-scan`. The fan-in is `needs: [package-review, risk-scan]`, which waits for every matrix leg and for the scan. `if: ${{ always() }}` lets the merger run and report even when a leg failed, which is what you want when partial results are still useful. Remove it if a merged plan built from incomplete inputs would be dangerous. The workflow-level `concurrency` group stops two whole runs for the same ref from overlapping.

Copilot CLI, the Copilot SDK and cloud agent

- **`/fleet` in Copilot CLI.** The main agent analyses the prompt, decides whether it can be split into subtasks, acts as orchestrator, and runs subagents in parallel where possible, each with its own context window. The documented workflow is to build a plan in plan mode (`Shift+Tab`), then choose "Accept plan and continue in Autopilot execution mode + /fleet", or exit plan mode and prompt `/fleet implement the plan`. You can name a custom agent in the `/fleet` prompt for a specific subtask. The `--fleet` option starts fleet mode from the command line together with `-i`, `-p` or piped input, and cannot be combined with `--acp`.
- **Copilot SDK.** Pass `customAgents` when creating a session; each agent needs at least `name` and `prompt`, and can set `description`, `tools`, `mcpServers`, `infer`, `skills`, `model` and `reasoningEffort`. The runtime matches the prompt against each agent's name and description and delegates to a sub-agent unless `infer` is `false`. Sub-agents do not inherit skills from the parent.
- **Copilot cloud agent** (formerly Copilot coding agent). Each task runs in its own ephemeral GitHub Actions environment and produces its own pull request. Parallelism here means several tasks, each on its own `copilot/` branch. You can start tasks from the CLI with `gh agent-task create`, including `--custom-agent` to choose an agent profile and `--base` to choose a base branch. PREVIEW

Listing 5.4: Starting parallel work from the terminal

```
# Local, parallel subagents (Copilot CLI fleet mode, non-interactive)
copilot --fleet -p "Add unit tests for every module in src/" --allow-tool write

# Interactive: plan first, then fan out
#   Shift+Tab           switch to plan mode and agree the plan
#   /fleet implement the plan. Use the @test-writer agent for the tests.
#   /tasks              list subagent tasks (Enter = details, x = kill, r = remove)

# Remote, one cloud agent task (and one pull request) per item
for svc in api web worker; do
  gh agent-task create "Upgrade the logging library in services/$svc only" \
    --custom-agent dependency-upgrader --base main
done
```

Control	VS Code custom agents	GitHub.com cloud agent profile	Copilot CLI agent file	Copilot SDK
Which subagents a coordinator may call	`agents` plus the `agent` tool	`agent` tool alias (`custom-agent`, `Task`)	Task tool	Session `customAgents`
Hide from the picker	`user-invocable: false`	`user-invocable: false`	Not in the CLI front matter reference	Not applicable
Block automatic selection or delegation	`disable-model-invocation: true` (`infer` deprecated)	`disable-model-invocation: true` (`infer` retired)	`infer: false`	`infer: false`
Guided stage transitions	`handoffs`	Ignored	Not in the CLI front matter reference	Not applicable

The property names differ between surfaces, as the table shows. Answer an exam question using the surface that the scenario names.

5.1.2 Configure agent isolation for parallel execution

Parallel agents are only safe when they cannot overwrite each other. Isolation works at five levels, and a robust design usually combines several of them.

Isolation mechanism	What it separates	How to configure it	What it does not do
Separate branches	Commits and history of each agent's work	Cloud agent creates a new `copilot/` branch per task and can push only to that branch (or to the PR branch when mentioned on an existing PR). Locally, one branch per agent session.	Does not stop two branches from editing the same file; the conflict appears at merge time.
Git worktrees	Working files and uncommitted changes on one machine	VS Code Agents window: **New Worktree** isolation. Copilot CLI: `--worktree`. Plain Git: `git worktree add`.	A worktree does not restrict commands, network access, or access to files outside the worktree (VS Code docs).
Separate pull requests	Review, checks and merge decision per unit of work	One cloud agent task, issue assignment or `gh agent-task create` per unit. Each produces its own draft PR.	PRs can still overlap in files; you need an overlap check (5.1.3).
File ownership	Which agent may change which paths	Instruct each agent on its scope; restrict tools (read-only workers); use path-specific `*.instructions.md` with `applyTo`; CODEOWNERS plus required code owner review for shared paths.	Prompt-level scope is guidance, not enforcement; pair it with review and checks.
Concurrency groups	Workflow runs or jobs that touch a shared resource	`concurrency.group` at workflow or job level; `cancel-in-progress`; `queue`.	Serialises runs in the same group; it does not merge or reconcile their outputs.

Branches and pull requests

The cloud agent's branch model is itself an isolation control. GitHub Docs (risks and mitigations) state that the cloud agent can push to a single branch only: a new `copilot/` branch for a new task, or the existing pull request's branch when you mention `@copilot` on that PR. It cannot run `git push` directly, it is subject to branch protections and required checks, it cannot mark its draft PR ready for review, and it cannot approve or merge. So "one task, one branch, one PR" is the default unit of isolation for remote agents. When two cloud agent tasks need to touch the same area, sequence them (start the second from the merged result) rather than running them together.

Worktrees for local parallel sessions

A Git repository normally has one working directory, the primary worktree. A linked worktree is another working directory for the same repository: worktrees share history but not working files or uncommitted changes, and Git prevents the same local branch from being checked out in more than one worktree at once. That last rule is useful: two agents cannot accidentally work on the same branch in two worktrees.

In the VS Code Agents window, choose **New Worktree** when you start a session and pick a base branch. The docs recommend a worktree for parallel tasks that should not modify your active workspace (it starts from committed Git state and you integrate the result afterwards) and **Folder** for small interactive tasks that should use your current files and uncommitted changes. In Copilot CLI, `-w` or `--worktree[=NAME]` creates or reuses an isolated worktree under `<repo>.worktrees/` by default and starts the session inside it.

Listing 5.5: Worktree isolation for two local agents

```
# Copilot CLI: each session gets its own worktree and branch
copilot --worktree=agent-api   -i "Refactor the API error handling in services/api"
copilot --worktree=agent-infra -i "Update Terraform modules under infra/ only"
```

```
# Equivalent with plain Git (one branch per agent, created from main)
git worktree add -b agent/api   ../repo-agent-api   main
git worktree add -b agent/infra ../repo-agent-infra main
git worktree list

# After each agent's branch is pushed and its PR is merged
git worktree remove ../repo-agent-api
git worktree remove ../repo-agent-infra
git worktree prune
```

PRODUCTION NOTE

A worktree isolates files, not privileges. An agent in a worktree can still run any command its permission level allows and can reach paths outside the worktree. Combine worktrees with tool restrictions and approval settings when the agent runs shell commands.

File ownership

Give each parallel agent a disjoint set of paths and make that ownership visible in three places: the agent's instructions ("change files under `infra/` only"), its tools (reviewers and researchers get `read` and `search` only), and the repository's controls. A CODEOWNERS file in `.github/`, the root or `docs/` assigns owners per path, and when code owner review is required, any PR touching a shared file such as `shared/config.yaml` needs that owner's approval. That adds a human gate, but it does not detect overlap by itself, so pair it with the automated check in Listing 5.7.

Concurrency groups

`concurrency` ensures that at most one job or workflow in the same group runs at a time. Key behaviours from GitHub Docs:

- It can be set at workflow level (whole runs) or job level (`jobs.<job_id>.concurrency`). Steps cannot declare concurrency.
- The group can be any string or expression using the `github`, `inputs`, `vars`, `needs`, `strategy` and `matrix` contexts. Group names are case-insensitive.
- By default, a new queued run makes the existing *pending* run in the group be cancelled and replaced (`queue: single`). `cancel-in-progress: true` also cancels the *running* one.
- `queue: max` allows up to 100 pending runs in the group, processed in the order they started waiting. It cannot be combined with `cancel-in-progress: true`; that combination fails validation.
- Include `github.workflow` in the group name so that two different workflows do not cancel each other.

Listing 5.6: Serialise runs per pull request, keep different pull requests parallel

```
# .github/workflows/agent-pr-validation.yml
on:
  pull_request:

concurrency:
  # For pull_request events github.ref is refs/pull/<number>/merge,
  # so each PR gets its own group; different PRs still run side by side.
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true        # newer push from the agent supersedes the old run

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: npm ci && npm test

# A different workflow that applies a shared resource must not drop runs:
#
# concurrency:
```

```
#   group: shared-config-apply
#   queue: max                    # queue every run; no cancel-in-progress here
```

COMMON TRAP

A single fixed group (for example `group: all-agents`) shared by two agents' workflows serialises them. That avoids collisions, but it breaks any requirement that the agents run concurrently. Use a per-branch or per-PR key to keep parallelism, and use a fixed key only for a genuinely shared resource such as a deployment environment.

5.1.3 Detect and resolve agent conflicts

The outline names three kinds of conflict. Each has a different detection signal and a different fix, and the exam often gives you the symptom and asks for the control.

Conflict type	Typical symptom	Detection	Resolution	Prevention
Overlapping code changes	Two agent PRs edit the same file; merge conflicts or a later PR silently overwrites the first one's intent.	Required status check that compares changed files across open agent PRs; merge conflict status on the PR; CODEOWNERS review requests on shared paths.	Merge one PR, then rebase or re-run the second agent on the updated base; or consolidate both changes into one PR owned by one agent.	Disjoint file ownership; separate branches and worktrees; sequence agents that share a file.
Duplicated effort	Two agents (or two sessions) solve the same task; two near-identical PRs appear.	Session list on the agents page; `gh agent-task list`; open PRs linked to the same issue; the coordinator's task list.	Keep the better result, close the duplicate PR, stop or archive the extra session.	One orchestrator owns the task breakdown; one issue per unit of work; assign each issue once; label agent PRs.
Contradictory outputs	The security reviewer says "block" and the style reviewer says "approve"; two research subagents recommend different libraries.	Aggregator compares structured outputs; the coordinator is instructed to list disagreements explicitly.	Apply a declared resolution rule: precedence (security outranks style), vote or majority for classification, weighted merge for scored recommendations, or escalate to a human.	Give each agent a distinct, non-overlapping remit and a structured output format so results can be compared.

The Microsoft Learn architecture guide notes that concurrent orchestration "requires conflict resolution when results contradict", and lists aggregation strategies: voting or majority rule for classification, weighted merging for scored recommendations, or a model-synthesised summary to reconcile results. It also lists "sharing mutable state between concurrent agents" as an antipattern. Two agents writing the same configuration file is exactly that antipattern.

Listing 5.7: A required status check that fails when another open agent PR touches the same files

```
# .github/workflows/agent-overlap-check.yml
name: agent-overlap-check
on:
  pull_request:
    types: [opened, synchronize, reopened, labeled]

permissions:
  contents: read
  pull-requests: read

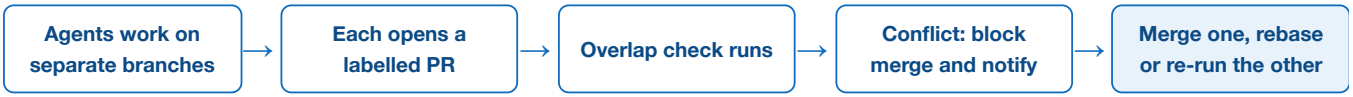
jobs:
  overlap:
    if: contains(github.event.pull_request.labels.*.name, 'agent-work')
    runs-on: ubuntu-latest
    steps:
```

```
      - name: Compare changed files with other open agent PRs
        env:
          GH_TOKEN: ${{ github.token }}
          GH_REPO: ${{ github.repository }}
          PR: ${{ github.event.pull_request.number }}
        run: |
          gh pr view "$PR" --json files --jq '.files[].path' | sort -u > mine.txt
          gh pr list --state open --label agent-work --json number,files \
            --jq ".[] | select(.number != $PR) | .number as \$n | .files[] | \"\(\$n) \(.path)\"" \
            > others.txt
          overlap=$(awk 'NR==FNR { mine[$1]=1; next } ($2 in mine)' mine.txt others.txt)
          if [ -n "$overlap" ]; then
            echo "::error::Files also changed by other open agent PRs:"
            echo "$overlap"
            exit 1
          fi
```

Add the `overlap` job as a required status check in branch protection or a ruleset, and label each agent PR `agent-work` (for example from the agent's instructions or from the workflow that starts it). The agents still run in parallel on isolated branches; the check only stops a conflicting PR from merging until a human or the coordinator resolves it. That combination (parallel execution, branch isolation, pre-merge detection) is usually the best answer when a scenario requires all three.

EXAM TIP

When the requirement is "detect file-level conflicts before merge" look for a **required status check**. CODEOWNERS adds a reviewer, read-only tools stop the agent working, and a shared concurrency group serialises the agents. None of those three detects overlap.



5.2 Configure observability for multi-agent behavior

With one agent you can watch a single session log. With several, you need to answer harder questions afterwards: which agent made this change, what was it told, what did it hand to the next agent, and why did the coordinator choose this result over another. GitHub's approach is to keep that evidence inside standard development tooling (pull requests, commits, session logs, workflow logs and artifacts, the audit log) rather than in a separate system.

5.2.1 Configure multi-agent workflows to produce artifacts suitable for review and audit

An artifact is suitable for review and audit when it is durable, attributable to an agent and a person, linked to the change it explains, and readable by the people who approve the change. The table lists what each GitHub surface produces by default and what you add yourself.

Artifact	Produced by	Audit value
Draft pull request with description	Cloud agent, one per task	Unit of review. When a custom agent did the work, the PR description notes which custom agent was used.
Commits authored by Copilot, co-authored by the requester	Cloud agent	Identifies agent-generated code and who started the task. Commits are signed and show as Verified.
Session log link in each commit message	Cloud agent	Traces every commit to the reasoning and tool calls that produced it, for code review or audit.

Artifact	Produced by	Audit value
Session logs and overview (progress, token usage, session length)	Agents page and agents panel on GitHub	Shows internal reasoning, tools used, tests and linters run, and security validation results.
Audit log events	Organisation and enterprise audit log; filter `actor:Copilot`	Administrative trail of agent activity; can be streamed (Preview for some enterprise types).
Workflow logs, job summaries and uploaded artifacts	GitHub Actions	Per-job record of each agent stage, with a Markdown summary on the run page (`$GITHUB_STEP_SUMMARY`) and downloadable files with a `retention-days` setting.
Sub-agent lifecycle events	Copilot SDK session events	`subagent.selected` , `subagent.started` , `subagent.completed` , `subagent.failed` , `subagent.deselected` , with `toolCallId` , duration, tokens and tool call counts.
Exported session	Copilot CLI `/share` or `--share` / `--share-gist`	Markdown, HTML or gist export of a local session for attaching to a PR or incident.

Two configuration choices make multi-agent artifacts much easier to audit:

1. **Make every stage emit a named artifact.** In Actions, upload one artifact per agent and per item (`review-api` , `review-web` , `risk`) as in Listing 5.3, and write the merged result to the job summary. Set `retention-days` to match your audit policy; it cannot exceed the repository, organisation or enterprise limit.
2. **Make every agent produce structured output.** Instruct workers to return the same sections every time (findings, files touched, decisions, open questions). Structured outputs can be compared by the aggregator, diffed between runs, and read by reviewers without replaying the session.

> **NOTE**
> When building your own UI or log pipeline with the Copilot SDK, sub-agent events share the parent session's event stream. Sub-agent events carry an envelope-level `agentId` ; root agent and session-level events omit it. Use `toolCallId` to rebuild the execution tree (which sub-agent ran for which tool call, and whether it completed or failed).

5.2.2 Document key decisions, handoffs, and outcomes across agents

Session logs record what happened; they do not always say why a coordinator chose one path. Durable decision records fill that gap and also serve as the input for the next agent, which matters because a Local subagent does not inherit conversation history. Three places work well:

- **The plan file.** Have the planner write its plan to the repository (for example `docs/plans/<issue>.md`) and require later agents to record deviations from it. A plan that is a reviewed file can be validated and approved before any agent acts.
- **The handoff prompt.** A handoff `prompt` or a coordinator's subagent request is the contract between two agents. Write it to include goal, context, allowed actions and expected result, which is the structure the VS Code docs recommend for a self-contained delegated task.
- **The pull request.** The PR description and comments are where reviewers look. Ask the final agent to summarise which agents ran, what each concluded, which disagreements were resolved and how, and what remains open.

Listing 5.8: A handoff and decision record that each stage appends to, published to the run summary

```
<!-- docs/agent-runs/1423.md : written by the coordinator, one section per stage -->
# Agent run for issue #1423: rate limiting on public API

## Stage 1 - Planner (planner.agent.md @ 3f9c2e1)
```

```
- Input: issue #1423, ADR-012
- Output: docs/plans/1423.md (approved by @api-leads)
- Decision: token bucket in middleware, not per-route decorators

## Stage 2 - Implementer (implementer.agent.md @ 3f9c2e1)
- Files: src/middleware/rate_limit.py, tests/test_rate_limit.py
- Deviation from plan: none

## Stage 3 - Reviewers (fan-out)
| Reviewer          | Verdict | Key finding                          |
|-------------------|---------|--------------------------------------|
| security-reviewer | block   | limit key uses client-supplied header |
| style-reviewer    | approve | none                                 |

## Resolution
- Rule applied: security verdict takes precedence over style
- Action: Implementer re-run with finding; security-reviewer re-checked: approve
- Outcome: PR #1431 ready for human review
```

```
# In the workflow job that finishes the run
- name: Publish decision record
  run: cat "docs/agent-runs/${{ github.event.issue.number }}.md" >> "$GITHUB_STEP_SUMMARY"
- uses: actions/upload-artifact@v4
  with:
    name: agent-run-record
    path: docs/agent-runs/
    retention-days: 90
```

Recording the agent profile's commit SHA next to each stage (for example `planner.agent.md @ 3f9c2e1`) links every outcome to the exact agent definition that produced it. Custom agent versioning is based on the Git commit SHA of the profile file, so that SHA is a reliable version identifier (see 5.4.2).

EXAM TIP

"Document decisions and handoffs" answers point to durable, version-controlled artifacts: plan files, decision records, PR descriptions and job summaries. Chat history alone is not a durable record because it is not attached to the change and subagents do not share it.

5.2.3 Perform post-hoc analysis of multi-agent behavior

Post-hoc analysis reconstructs what happened after a run, usually to explain a bad outcome or to tune the workflow. Work from the change backwards and from the run forwards:



- **Session logs.** On the agents page, open a session to see the log and overview. Commits link to their session logs. Copilot Chat can answer "what changed, what was validated and why" for a completed session by drawing on its logs.
- **Session history queries.** You can ask Copilot in natural language about your synced sessions from Copilot CLI, cloud agent and code review, for example "Show me sessions where the agent struggled or I had to correct it multiple times." In Copilot CLI, `/chronicle` provides session history tools. You can only query sessions you started, unless a session has been shared.
- **CLI data.** `gh agent-task list --json` and `gh agent-task view --json` expose fields such as `state`, `createdAt`, `completedAt`, `pullRequestNumber` and `pullRequestState`. `gh run list --json` and `gh run view --log-failed` do the same for workflow runs.
- **Telemetry.** Enterprise owners can view recent agent sessions under **AI controls**, use the audit log, stream agent session events, and enable OpenTelemetry export from Copilot clients to analyse agent sessions centrally.

- **Per-agent metrics.** The Learn architecture guide recommends instrumenting all agent operations and handoffs, tracking performance and resource usage per agent, and monitoring token consumption per agent and per orchestration run. SDK events supply `durationMs`, `totalTokens` and `totalToolCalls` per sub-agent.

Listing 5.9: Post-hoc queries from the terminal

```
# Recent cloud agent sessions with their PRs and states
gh agent-task list -L 50 \
  --json id,name,state,createdAt,completedAt,pullRequestNumber,pullRequestState

# The full log of one session, by PR number
gh agent-task view --repo octo-org/payments 1431 --log

# Failed runs of the orchestration workflow, then only the failing steps' logs
gh run list --workflow agent-review-fanout --status failure \
  --json databaseId,headBranch,createdAt,conclusion
gh run view <run-id> --log-failed

# Job IDs for a run (needed for gh run rerun --job)
gh run view <run-id> --json jobs --jq '.jobs[] | {name, databaseId, conclusion}'
```

When you classify what went wrong, use the same root-cause categories as single-agent analysis (reasoning error, tool misuse, context or environment issue) and add the coordination categories that only appear with several agents: a bad task split, missing context in a delegation, two agents given overlapping scope, an aggregation rule that picked the wrong result, or a timing problem such as a superseded run being cancelled.

5.3 Detect and respond to multi-agent failures and degraded behavior

Multi-agent systems fail like distributed systems: one node fails, a message is lost, or an error cascades. The Learn architecture guide lists the expected mitigations: timeouts and retries, graceful degradation when an agent faults, surfacing errors instead of hiding them, validating each agent's output before passing it on, circuit breakers for agent dependencies, avoiding shared single points of failure, and checkpoints to resume an interrupted orchestration. This section maps those ideas to GitHub signals and controls.

5.3.1 Identify failed, partial, or stalled agent executions

State	What it looks like	Where to see it
Failed	The agent or job ended with an error.	Actions run conclusion `failure` or `timed_out`; SDK `subagent.failed` event with `error`; failed task in the CLI `/tasks` list; `gh run watch --exit-status` returns non-zero.
Partial	Some agents finished and others did not, or one agent completed only part of its task.	Matrix with some legs failed or cancelled; fan-in job skipped because a needed job failed; PR that lists unresolved items; session summary that reports open work; draft PR with failing checks.
Stalled	No visible progress for a long time.	Session log with no new activity (watch it live with `gh agent-task view <id> --log --follow`); job running close to `timeout-minutes`; run sitting in `pending` behind a concurrency group; PR waiting for **Approve and run workflows**.

Facts worth memorising:

- **Cloud agent stall timeout.** GitHub Docs (troubleshooting) say Copilot can appear stuck for a while and then continue, but if the session remains stuck it times out after an hour. You can retry by unassigning the issue and reassigning it to Copilot. If it got stuck while responding to a comment, add the same comment to the PR again.
- **Job timeout.** `jobs.<job_id>.timeout-minutes` defaults to 360 minutes. Set a smaller value on agent jobs so a stalled agent fails fast instead of holding a runner and a concurrency slot for six hours.

- **Workflows waiting for approval.** By default, workflows do not run on cloud agent pushes until a user with write access clicks **Approve and run workflows**. A PR with no checks is often waiting for that, not stalled.
- **Pending is not running.** A run in a concurrency group waits in `pending`. With the default queue behaviour, a newer run replaces the older pending one, so a run can disappear from the queue as cancelled without having failed.
- **Skipped downstream jobs.** If a job in a `needs` chain fails or is skipped, every job that needs it is skipped unless it uses a condition such as `always()`. A "skipped" fan-in is a symptom of an upstream failure.

Listing 5.10: Watching agent sessions and runs live

```
# Cloud agent sessions (GitHub CLI 2.80.0 or later; public preview)
gh agent-task list                                  # recent sessions and their state
gh agent-task view 1431 --log --follow              # stream the live session log
gh agent-task create "Fix flaky test in tests/api" --follow   # start and stream

# GitHub Actions runs that orchestrate agents
gh run list --workflow agent-review-fanout --status in_progress
gh run watch <run-id> --compact --exit-status && echo "all agent jobs passed"

# Local Copilot CLI session
#   /tasks   list subagents and shells; Enter = details, x = kill, r = remove
```

COMMON TRAP

`--log` shows a session log once; `--follow` streams it as the agent works. `--jq` and `--json` filter output but never stream, and `--web` only opens the browser. For workflow runs, the streaming equivalent is `gh run watch`, and `--exit-status` makes it return non-zero when the run fails, which is what you need in a script.

5.3.2 Respond to degraded behavior or coordination across agents

Degraded behaviour is worse than a clean failure because it looks like progress: an agent loops, drifts from the plan, ignores a constraint, or keeps passing low-quality output downstream. Respond in increasing order of force.

Symptom	First response	If that does not work
One agent heading in the wrong direction	**Steer** it. On GitHub, type a follow-up in the session's prompt box; Copilot applies it after the current tool call. In Copilot CLI, a prompt sent while Copilot is thinking is treated as steering; `Ctrl+Enter` queues it instead. In `/tasks` you can open a subagent and send it a steering message.	**Stop** the session (**Stop session** ends the Actions run and keeps commits already pushed) or kill the subagent (x in `/tasks`).
Low-quality output passed to the next stage	Validate output before passing it on: the coordinator or a check rejects malformed or off-topic results and retries with clarified context.	Halt the pipeline; make the gate a required check or a human handoff with `send: false`.
One matrix leg keeps failing	Set `fail-fast: false` so healthy legs finish; use `continue-on-error` for legs that are allowed to fail.	Exclude the item, fix its input, re-run only failed jobs.
Agents overload a shared dependency (rate limits, shared endpoint)	Reduce parallelism with `max-parallel` or a job-level concurrency group.	Circuit-break: stop dispatching new agents until the dependency recovers.
Coordination loop (agents repeatedly re-delegate or hand back)	Add a stopping condition and a maximum number of rounds to the coordinator's instructions; keep nesting disabled unless needed.	Replace dynamic delegation with a fixed sequential or fan-out workflow for that task.
Superseded runs pile up	`cancel-in-progress: true` on a per-PR group.	`gh run cancel <run-id>` for stragglers.

PRODUCTION NOTE

Steering messages on GitHub consume AI credits, and steering is not available for third-party coding agents. If you find yourself steering the same correction repeatedly, put it in the agent profile or in custom instructions instead (see 5.4.2).

5.3.3 Implement multi-agent recovery patterns, including rollback and human-in-the-loop

Retry and resume

- **Re-run only what failed.** `gh run rerun <run-id> --failed` re-runs failed jobs including their dependencies; `--job <databaseId>` re-runs one job (use the `databaseId` from `gh run view --json jobs`, not the number in the browser URL); `--debug` adds debug logging.
- **Retry a stuck cloud agent task.** Unassign and reassign the issue, or repeat the comment on the PR.
- **Resume from a checkpoint.** Persist plans, decision records and intermediate artifacts (5.2.2) so a restarted agent continues from the last good stage instead of replaying earlier ones. Copilot CLI can resume sessions with `--resume` or `--continue`.

Rollback

What to undo	Mechanism
A local agent's last turn, before commit	Copilot CLI `/undo` or `/rewind`: roll back the conversation only, or conversation and files (Copilot tracks file changes per turn, including sub-agents, so Git is not required).
An agent's unmerged work	Close the PR and delete the branch, or remove the worktree. Nothing reached the default branch.
A merged agent PR	**Revert** button on the merged PR, or `gh pr revert <number>`. Both create a new PR that reverts the merge; you need write permission. Revert individual commits with `git revert` if the PR revert conflicts.
A bad agent profile change	`git revert` the commit that changed the `.agent.md` file, through a reviewed PR. New tasks use the reverted profile once it is on the branch (5.4.2).

Listing 5.11: Recovery commands

```
# Re-run only the failed agent jobs of a run, with debug logging
gh run rerun <run-id> --failed --debug

# Revert a merged agent PR (creates a new revert PR)
gh pr revert 1431 --title "Revert agent change from #1431" \
  --body "Rolled back after the fan-in reviewer flagged a regression in staging."

# Revert a single bad commit instead, on a branch, then open a PR
git switch -c revert-1431 origin/main
git revert --no-edit <sha>
git push -u origin revert-1431
gh pr create --fill
```

Human-in-the-loop

Human-in-the-loop (HITL) recovery means a person decides at the point where an agent's judgement is not enough. GitHub builds several mandatory gates into the cloud agent, and you can add your own:

- Cloud agent PRs are drafts that a human must review and merge; the agent cannot mark them ready, approve or merge them.
- The user who asked Copilot to create a PR cannot approve it, preserving required approvals. When the cloud agent opens a PR under its own app identity, one extra approval is required if the repository already requires at least one.
- Workflows do not run on agent pushes until a person clicks **Approve and run workflows** (unless you configure automatic runs).

- Handoffs with `send: false` pause between stages until the developer submits the prompt.
- Required status checks (such as Listing 5.7) and CODEOWNERS review stop conflicting or sensitive changes from merging.

The Learn guide adds two design rules for HITL gates: persist state at the gate so the workflow can resume without replaying earlier agents, and scope gates to the sensitive operations (specific tool calls or paths) so low-risk work continues autonomously. That is how you keep delivery fast while still keeping a human on the decisions that matter.



5.4 Manage the lifecycle of agents within multi-agent workflows

Agent profiles are code. They live in Git (`.github/agents/` in a repository, `agents/` in an organisation's or enterprise's `.github-private` repository, or `.github` for organisations, and `~/.copilot/agents` for a user), so their lifecycle follows the same practices as any other shared configuration: propose by pull request, review, merge, version by commit, roll back by revert, and keep history when you remove them.

Level	Location of agent profiles	Scope
User	`~/.copilot/agents/`	You, across repositories
Repository	`.github/agents/`	That repository
Organisation PREVIEW	`agents/` in the organisation's `.github` or `.github-private` repository	All organisation members
Enterprise PREVIEW	`agents/` in the `.github-private` repository of the organisation designated in enterprise settings	All organisations in the enterprise

On naming conflicts, the lowest level wins: a repository-level agent overrides an organisation-level agent, which overrides an enterprise-level agent. The configuration file name (without `.md` or `.agent.md`) is what is used for deduplication between levels. File names may contain only `.`, `-`, `_`, `a-z`, `A-Z` and `0-9`.

5.4.1 Add agents to existing multi-agent workflows

Adding an agent touches two things: the new agent's own profile, and whatever routes work to it. Follow this sequence:

1. **Define a narrow remit.** Write a specific `description` (required on GitHub.com) because both VS Code and the Copilot runtime use it to decide when to delegate. Restrict `tools` to what the remit needs; omitting `tools` grants all tools, and `tools: []` disables all.
2. **Decide how it is reached.** A worker that should only be called by a coordinator gets `user-invocable: false`. If it must never be chosen automatically, set `disable-model-invocation: true` and list it explicitly in the coordinators that may use it.
3. **Wire it into the workflow.** Add its exact, case-sensitive `name` to the coordinator's `agents` list (and confirm the coordinator has the `agent` tool), add a `handoffs` entry if it is a stage in a VS Code sequence, add an Actions job with the right `needs` if it is a pipeline stage, or name it in `/fleet` prompts.
4. **Test before release.** For organisation or enterprise agents, GitHub documents a test-then-release path: create the profile in `.github/agents` inside the `.github-private` repository (available only to members with access to that repository, and only for tasks in that repository), test it from the agents tab, then move the file to the `agents` directory and merge to make it available across the organisation or enterprise.
5. **Ship it by pull request.** Agents on GitHub.com become available once the profile is merged into the default branch.

COMMON TRAP

A coordinator with an explicit `agents` list will not call a new agent until you add the new agent's name to that list. Conversely, if the coordinator omits `agents` (or uses `['*']`), every new custom agent that is not blocked by `disable-model-invocation` becomes a potential subagent the moment it is merged. In a production workflow, prefer an explicit list so that adding an agent is a deliberate, reviewed change to the coordinator too.

5.4.2 Update, reconfigure, or replace agents without disrupting active workflows

The key fact comes from the GitHub Docs custom agents configuration reference: **custom agent versioning is based on Git commit SHAs for the agent profile file**. You can create branches or tags with different versions of an agent. When you assign a custom agent to a task, it is instantiated from the latest version of the profile for that repository and branch. When the agent creates a pull request, interactions within that PR use the same version of the custom agent for consistency.

The consequences for active workflows:

- An agent already working on a PR keeps the profile version it started with, so merging an updated profile does not change the behaviour of in-flight PR work mid-stream.
- New tasks pick up the updated profile from the branch they target. Changing a profile on a feature branch lets you trial it without affecting tasks based on the default branch.
- Rolling back is a Git operation: revert the commit that changed the profile, and the next task uses the previous behaviour.

Review agent profile changes like code

Because a changed profile changes what every future task does, profile edits deserve the same review as production code:

- **CODEOWNERS** on `.github/agents/` with required code owner review routes every profile change to the team that owns the agents. For code owners to be requested, the CODEOWNERS file must be on the base branch of the PR.
- **Rulesets.** Enterprise owners can create a ruleset from **AI controls** (Agents tab, "Protect agent files using rulesets") that allows only enterprise owners to edit agent profiles. Members with write access can still open PRs proposing changes; members with bypass access merge them.
- **Validation checks** on PRs that touch agent profiles, as in Listing 5.12.

Listing 5.12: CODEOWNERS for agent profiles and a validation check on profile changes

```
# .github/CODEOWNERS
/.github/agents/        @octo-org/agent-platform
/.github/workflows/     @octo-org/agent-platform

# .github/workflows/agent-profile-check.yml
name: agent-profile-check
on:
  pull_request:
    paths:
      - '.github/agents/**'
permissions:
  contents: read
jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Every profile has a description and an explicit tools list
        run: |
          status=0
          for f in .github/agents/*.md; do
            grep -q '^description:' "$f" || { echo "::error file=$f::missing description"; status=1; }
            grep -q '^tools:'       "$f" || { echo "::error file=$f::tools must be explicit"; status=1; }
```

```
          done
          exit $status
```

Reconfigure versus replace

Change	Approach	Why it avoids disruption
Small instruction or tool tweak	Edit the profile in a PR; merge to the default branch.	In-flight PRs keep their version; new tasks get the new one; one revert undoes it.
Model change	In IDE and CLI profiles, change `model`; VS Code also accepts a prioritised list so the next model is tried if the first is unavailable.	A fallback list keeps the agent working if a model is unavailable.
Behaviour change you want to trial	Change the profile on a branch, or create a test copy under `.github/agents` in `.github-private` before moving it to `agents/`.	Only tasks on that branch or repository use the trial version.
Full replacement	Add the new agent under a new file name alongside the old one; switch coordinators' `agents` lists and handoffs to the new name in one reviewed PR; retire the old one later (5.4.3).	Old and new run side by side; the cut-over is one revertible commit.

Listing 5.13: Find and revert the commit that changed an agent profile

```
# Which commits changed the reviewer profile, and what changed
git log --oneline -- .github/agents/reviewer.agent.md
git show 7d41a0c -- .github/agents/reviewer.agent.md

# Revert that one commit on a branch and propose it for review
git switch -c revert-reviewer-profile origin/main
git revert --no-edit 7d41a0c
git push -u origin revert-reviewer-profile
gh pr create --title "Revert reviewer agent profile to previous version" \
  --body "Reverts 7d41a0c. New reviewer runs flagged every PR as blocking."

# Pin a known-good version for later comparison or rollback
git tag agents/reviewer-v3 7d41a0c~1
git push origin agents/reviewer-v3
```

EXAM TIP

If a deployment or validation step must use exactly the reverted profile, check out and refer to the immutable commit (`${{ github.sha }}`), not a moving ref such as a branch name. A branch name can point to a different commit by the time the step runs; a commit SHA cannot.

COMMON TRAP

A repository-level profile with the same file name as an organisation-level agent overrides the organisation version for that repository. That is useful for a local fix, but it also means a forgotten repository copy can silently shadow an updated organisation agent. Audit for duplicate file names when you update shared agents.

5.4.3 Retire agents while preserving auditability and workflow continuity

Retiring an agent has two goals that pull in different directions: stop new work from reaching it, and keep every trace of the work it has already done. Do it in stages.



1. **Stop new routing.** Remove the agent from every coordinator's `agents` list, from `handoffs`, from Actions jobs and from automation prompts, pointing them at the replacement if there is one.
2. **Hide and block.** Set `user-invocable: false` so nobody picks it, and `disable-model-invocation: true` so it is not chosen automatically. Remember that an explicit listing in some coordinator's `agents` overrides `disable-model-invocation` in VS Code, so step 1 matters.
3. **Let active work finish.** PRs the agent already opened keep using the profile version they started with. Review and merge or close them before you delete the profile.
4. **Delete through a pull request.** Deleting a shared workspace agent removes it for everyone when they receive the change, so make it a reviewed PR with a description that explains the retirement and names the replacement. Do not "park" the file in a subfolder of `.github/agents`: VS Code treats any `.md` file in that folder as a custom agent.
5. **Preserve the record.** Git history keeps every version of the profile (a tag on the last version makes it easy to find). Cloud agent sessions can be archived but not deleted. Commits remain linked to their session logs, signed and co-authored, and audit log events remain. Together these let an auditor reconstruct what the retired agent did and under which definition.

Listing 5.14: Deprecation stage of a retiring agent profile

```
# .github/agents/legacy-migrator.agent.md   (deprecation PR, before deletion)
---
name: Legacy Migrator
description: DEPRECATED. Replaced by schema-migrator. Do not use for new work.
user-invocable: false
disable-model-invocation: true
tools: ['read', 'search']
---
This agent is retired. Its last active version is tagged agents/legacy-migrator-final.
Use the schema-migrator agent for new database migration tasks.
```

> **NOTE**
> Reducing the retiring agent's `tools` to read-only, as in Listing 5.14, limits harm if something still calls it during the transition. It is a safety net, not a replacement for removing it from coordinators.

5.5 Key terms

Orchestrator (coordinator)	The agent that holds the overall goal, splits it into tasks, delegates them and combines the results.
Subagent	An agent invoked by another agent to complete a delegated task in its own context; it returns a result to the caller. In VS Code Local sessions it is started by `runSubagent` (the `agent` tool set).
`agents` property	VS Code custom agent front matter listing which agents may run as subagents: named list, `['*']` or omitted for all, `[]` for none. Requires the `agent` tool.
Handoff	A VS Code front matter entry (`label`, `agent`, `prompt`, `send`, `model`) that shows a button to switch to another agent with a pre-filled prompt. Ignored by cloud agent on GitHub.com.
`user-invocable` / `disable-model-invocation`	Separate controls for picker visibility and automatic or subagent invocation. They replace the deprecated `infer` in VS Code and the retired `infer` on GitHub.com.
Fan-out/fan-in	Parallel producers followed by an aggregator; in Actions, independent jobs plus a job with `needs` on all of them.
Fleet mode	Copilot CLI orchestration (`/fleet` or `--fleet`) in which the main agent runs subagents in parallel where possible.
Linked worktree	An extra working directory for the same repository with its own checked-out branch; isolates files and uncommitted changes, not permissions.

Concurrency group	An Actions key that allows at most one running job or workflow per group, with `cancel-in-progress` and `queue` controlling what happens to the others.
Agent profile version	The Git commit SHA of the `.agent.md` file used to instantiate a custom agent; PR interactions keep the version the PR started with.

5.6 Additional practice questions

Practice 5.1 MULTIPLE CHOICE

A release team uses VS Code. `release-coordinator.agent.md` has `tools: ['agent', 'read', 'search']` and `agents: []`. Two workers, `Changelog Writer` and `Version Checker`, exist in `.github/agents/` with default settings. The coordinator never delegates. You need the coordinator to call those two workers and no other agents. What should you do?

A. Remove the `agents` property from the coordinator.
B. Set `agents: ['Changelog Writer', 'Version Checker']` in the coordinator.
C. Set `user-invocable: true` in both workers.
D. Add a `handoffs` entry for each worker to the coordinator.

Practice 5.2 MULTIPLE CHOICE

You create a helper agent named `Schema Inspector`. It must not appear in the agents dropdown, but the `Data Coordinator` agent must still be able to run it as a subagent. Which front matter setting should you add to `Schema Inspector`?

A. `disable-model-invocation: true`
B. `user-invocable: false`
C. `infer: false`
D. `target: github-copilot`

Practice 5.3 MULTIPLE CHOICE

In VS Code your team uses `planner.agent.md`, which has a handoff with `send: true` to an implementer agent. A product owner selects the same planner as the custom agent for an issue assigned to Copilot cloud agent on GitHub.com and expects implementation to start automatically after planning. What happens?

A. The handoff runs automatically on GitHub.com because `send` is `true`.
B. The handoff is ignored on GitHub.com; the planner completes the task on its own, so later stages must be separate tasks or workflow steps.
C. The cloud agent rejects the profile because it contains an unsupported property.
D. The cloud agent converts the handoff into a subagent call to the implementer.

Practice 5.4 MULTIPLE CHOICE

A workflow deploys to a shared staging environment on every push to `main`. Several agent PRs are often merged within minutes of each other. Only one deployment may run at a time, and every merge must be deployed in order: no run may be dropped. Which `concurrency` configuration should you use?

A. `group: staging-deploy` and `cancel-in-progress: true`
B. `group: staging-deploy` and `queue: max`
C. `group: ${{ github.run_id }}`
D. `group: staging-deploy`, `queue: max` and `cancel-in-progress: true`

Practice 5.5 MULTIPLE SELECT (CHOOSE TWO)

A workflow runs a matrix of 12 per-service agent jobs, followed by a `summarise` job with `needs` on the matrix job. When one service's agent fails, the other legs are cancelled and `summarise` is skipped. You need every service to finish and the summary to report partial results. Which two changes should you make?

A. Set `strategy.fail-fast: false` on the matrix job.
B. Add `if: ${{ always() }}` to the `summarise` job.
C. Set `strategy.max-parallel: 1` on the matrix job.
D. Set `continue-on-error: true` on the `summarise` job.
E. Add `cancel-in-progress: false` to a workflow concurrency group.

Practice 5.6 MULTIPLE CHOICE

A developer wants to run two Copilot CLI sessions at the same time in one local clone: one rewriting documentation, one refactoring the API. Neither session may see or overwrite the other's uncommitted edits, and each result must be reviewable as its own branch. What should the developer do?

A. Start each session with `copilot --worktree=<name>`.
B. Start both sessions with `copilot --fleet`.
C. Run both sessions in the same folder but with different custom agents.
D. Use `/tasks` to pause one session while the other edits files.

Practice 5.7 MULTIPLE CHOICE

A Copilot cloud agent session working on an issue has shown no new session log activity for 20 minutes. You need to apply the documented behaviour and response. Which statement is correct?

A. The session runs until someone stops it; close the pull request to free the runner.
B. Copilot can appear stuck and then continue; if it remains stuck it times out after an hour, and you can retry by unassigning and reassigning the issue to Copilot.
C. The session restarts itself every 10 minutes until it makes progress.
D. Run `gh run rerun --failed` on the session's workflow run to resume it.

Practice 5.8 MULTIPLE CHOICE

Six months after a Copilot cloud agent PR was merged, an auditor asks why a specific line was added and who started the task. You need the most direct evidence path. What should you use?

A. Search the organisation audit log for the text of the line.
B. Use `git blame` to find the commit; its message links to the agent session log, and the commit is authored by Copilot with the requester as co-author.
C. Ask Copilot cloud agent to regenerate the change and compare the result.
D. Download the workflow artifacts from the original run.

Practice 5.9 MULTIPLE CHOICE

You merge an update to `.github/agents/test-writer.agent.md` into `main` while three PRs opened earlier by the `test-writer` custom agent are still under review. A reviewer mentions `@copilot` on one of those PRs to request a change. Which version of the agent profile handles the request?

A. The new version on `main`.
B. The version the agent used when it created that pull request.
C. The organisation-level version of `test-writer`, if one exists.
D. None; Copilot ignores the mention until the PR is rebased on `main`.

Practice 5.10 MULTIPLE CHOICE

An organisation's platform team maintains an organisation-level `security-reviewer` agent in its `.github-private` repository. They want to trial an improved version privately, without changing the agent other members use, and then release it organisation-wide. What should they do?

A. Edit `agents/security-reviewer.md` directly on the default branch and ask members to report problems.
B. Create the test profile in `.github/agents` in the `.github-private` repository, test it from the agents tab, then move it into the `agents` directory and merge.
C. Copy the new profile into `.github/agents` in every repository in the organisation.
D. Set `user-invocable: false` on the organisation agent during the trial.

Answers and explanations

5.1: B. `agents: []` prevents any subagent use, and a named list permits only those agents. The coordinator already has the `agent` tool, which is required when `agents` is set. Removing the property (A) permits all available agents, which breaks "no other agents". `user-invocable` (C) only affects the picker. Handoffs (D) are user-selected transitions, not subagent permissions.

5.2: B. `user-invocable: false` hides the agent from the dropdown but leaves subagent invocation untouched. `disable-model-invocation: true` (A) does the opposite job (it blocks invocation by other agents unless explicitly listed). `infer` (C) is deprecated in VS Code. `target` (D) selects the environment.

5.3: B. The GitHub Docs configuration reference states that `handoffs` and `argument-hint` are not supported for Copilot cloud agent on GitHub.com and are ignored for compatibility. The profile is not rejected (C), and handoffs are never converted to subagent calls (D).

5.4: B. `queue: max` keeps up to 100 pending runs in the group and processes them in the order they started waiting, while the group still allows only one running deployment. A cancels the running deployment. C gives every run its own group, so nothing is serialised. D is a validation error because `queue: max` cannot be combined with `cancel-in-progress: true`. Without `queue: max`, a new run replaces the existing pending run.

5.5: A and B. `fail-fast` defaults to `true` and cancels in-progress and queued matrix jobs when one fails; setting it to `false` lets the other services finish. Jobs that `need` a failed job are skipped unless they use a condition such as `always()`. `max-parallel` (C) only limits concurrency. `continue-on-error` on the summariser (D) does nothing if the job is skipped. `cancel-in-progress` (E) is about separate runs, not matrix legs.

5.6: A. `--worktree` creates or reuses an isolated Git worktree (by default under `<repo>.worktrees/`) on its own branch and starts the session inside it. Worktrees do not share working files or uncommitted changes. `--fleet` (B) parallelises subagents inside one session in the same folder. Different agents in one folder (C) still share files. `/tasks` (D) manages subagents and shells; it is not an isolation mechanism.

5.7: B. This is the documented troubleshooting guidance for a cloud agent that appears stuck. Sessions do not run indefinitely (A) or restart themselves (C). `gh run rerun` (D) is for workflow runs; the documented retry for a stuck agent is to reassign the issue (or repeat the PR comment if it stalled on a comment).

5.8: B. Cloud agent commits are authored by Copilot with the person who started the task as co-author, are signed, and include a link to the session logs in the commit message, specifically for review and audit. The audit log (A) records events, not code lines. Regenerating (C) is not evidence. Artifacts (D) may already have passed their retention period.

5.9: B. Custom agent versions are Git commit SHAs of the profile. A task uses the latest version for its repository and branch when assigned, and once the agent creates a PR, interactions within that PR use the same version for consistency. New tasks will use the updated profile.

5.10: B. GitHub documents this test-then-release path: profiles in `.github/agents` inside `.github-private` are available only to members with access to that repository and only for tasks started there; moving the file to `agents/` and merging releases it. A changes the live agent for everyone. C creates many copies that each override the organisation agent. D hides the current agent from users without trialling anything.

Key takeaways

- Choose the simplest pattern that fits: sequential handoff for staged work with checkpoints, orchestrator with subagents for one owner with specialist helpers, fan-out/fan-in for parallel producers with an aggregator, and matrix or `/fleet` for parallel per-item work.
- A VS Code coordinator needs the `agent` tool and an `agents` list. `agents: []` means none, `['*']` or omitted means all, and an explicit listing overrides a worker's `disable-model-invocation: true`. Names are case-sensitive.
- `user-invocable` controls picker visibility; `disable-model-invocation` controls automatic or subagent invocation; `infer` is deprecated (VS Code) or retired (GitHub.com) but still appears in the Copilot CLI and SDK.
- Handoffs (`label` , `agent` , `prompt` , `send` , `model`) are VS Code only and are ignored by cloud agent on GitHub.com.
- Local subagents do not inherit chat history and cannot be sent follow-ups: delegate self-contained tasks with goal, context, allowed actions and expected result.
- Isolate parallel agents with separate branches and PRs, worktrees (`--worktree` , New Worktree, `git worktree add`), disjoint file ownership, and concurrency groups keyed per PR or branch.
- Detect overlapping changes with a required status check before merge; resolve contradictions with a declared precedence, vote or human escalation; prevent duplicated effort with one owner of the task breakdown.
- Cloud agent commits are authored by Copilot, co-authored by the requester, signed, and link to session logs. Add plan files, decision records, job summaries and named artifacts for multi-agent audit.
- Watch sessions with `gh agent-task view --log --follow` and runs with `gh run watch --exit-status` . A stuck cloud agent session times out after an hour; retry by reassigning.
- Recover with `gh run rerun --failed` , PR revert or `git revert` , CLI `/undo` , and human gates (draft PRs, Approve and run workflows, `send: false` handoffs, required checks).
- Agent profiles are versioned by Git commit SHA; PR interactions keep the version the PR started with. Review profile changes with CODEOWNERS and rulesets, and roll back with a revert.
- Retire agents in stages: remove routing, hide and block, let in-flight PRs finish, delete by reviewed PR, and rely on Git history, archived sessions, linked commits and the audit log for continuity.

References

- Microsoft Learn: Study guide for Exam GH-600: Developing in Agentic AI Systems
- Microsoft Learn: AI agent orchestration patterns (Azure Architecture Center)
- VS Code: Subagents
- VS Code: Custom agents in VS Code (front matter, handoffs, orchestration example)
- VS Code: Use the Agents window (folder versus worktree isolation)
- VS Code: Choose an agent harness
- VS Code: Git branches and worktrees in VS Code
- GitHub Docs: Custom agents and sub-agent orchestration (Copilot SDK)
- GitHub Docs: Delegating and parallelizing work with agents

- GitHub Docs: Speeding up task completion with the /fleet command
- GitHub Docs: GitHub Copilot CLI command reference
- GitHub Docs: Steering agents in GitHub Copilot CLI
- GitHub Docs: Using Copilot cloud agent from the GitHub CLI
- GitHub Docs: Managing agent sessions
- GitHub Docs: Troubleshooting GitHub Copilot cloud agent
- GitHub Docs: Risks and mitigations for Copilot cloud agent
- GitHub Docs: About custom agents
- GitHub Docs: Creating custom agents for Copilot cloud agent
- GitHub Docs: Custom agents configuration (properties, versioning, naming precedence)
- GitHub Docs: Preparing to use custom agents in your organization
- GitHub Docs: Preparing to use custom agents in your enterprise
- GitHub Docs: Testing and releasing custom agents in your organization or enterprise
- GitHub Docs: Monitoring agentic activity in your enterprise
- GitHub Docs: Control the concurrency of workflows and jobs
- GitHub Docs: Workflow syntax for GitHub Actions (needs, strategy, matrix, timeout-minutes)
- GitHub Docs: Passing information between jobs
- GitHub Docs: Storing and sharing data from a workflow (artifacts, retention-days)
- GitHub Docs: Workflow commands (GITHUB_STEP_SUMMARY)
- GitHub Docs: About code owners
- GitHub Docs: Reverting a pull request
- GitHub CLI manual: gh agent-task view, gh agent-task create, gh agent-task list
- GitHub CLI manual: gh run watch, gh run rerun, gh run view, gh pr revert, gh pr diff
- Git documentation: git-worktree

DOMAIN 6

Implement guardrails and accountability

Exam weight: 10–15%

This domain tests whether you can let agents move fast without letting them do damage. The exam expects you to reason about **which** agent actions are risky, **how much** autonomy each class of action deserves, and **which concrete GitHub control** enforces that decision: hooks, CLI deny rules, content exclusion, the agent firewall, push protection, `GITHUB_TOKEN` permissions, CODEOWNERS, rulesets, branch protection, deployment environments with required reviewers, workflow approval for agent pull requests, and the enterprise audit log.

Questions are usually scenario based: a log excerpt, a policy requirement or a compliance constraint, followed by "which control should you use?" The trap answers are almost always plausible controls that act at the wrong moment (after execution instead of before), on the wrong surface (an IDE setting for a cloud agent problem) or at the wrong layer (guidance in an instructions file instead of enforcement). Study this chapter by memorising **when** each control fires and **what it can and cannot stop**.

Official skill (Implement guardrails and accountability)	Covered in
Define autonomy levels: classify agent actions by operational, security and compliance risk to right-size human interventions	6.1.1, 6.1.2
Define autonomy levels: assign autonomy levels to maximise delivery speed while remaining compliant with organisational security and Responsible AI standards	6.1.3, 6.1.4
Guardrails and HITL: identify the subset of actions that require human judgment	6.2.1
Guardrails and HITL: block actions that violate defined security, compliance or Responsible AI policies	6.2.2, 6.2.3, 6.2.4
Guardrails and HITL: scope permissions and execution contexts to enforce least-privilege access	6.2.5
Guardrails and HITL: require explicit authorisation or controlled paths for irreversible or compliance-sensitive changes	6.2.6, 6.2.7
Guardrails and HITL: preserve execution velocity by minimising approvals that do not materially reduce risk	6.2.8
Accountability (supporting skill): audit logs, attribution of agent actions, retention through streaming	6.3.1, 6.3.2, 6.3.3

6.1 Define autonomy levels

6.1.1 Classify agent actions by operational, security and compliance risk

Autonomy is not a property of an agent; it is a property of an **action**. The same Copilot cloud agent (formerly Copilot coding agent) session might fix a typo in a README, add a dependency, edit `.github/workflows/deploy.yml` and propose a database migration. Each of those actions has a different blast radius, so each deserves a different level of human involvement. Microsoft's agent governance guidance makes the same point at the agent level: a single checklist applied to everything "over-governs simple agents" and "under-governs complex agents", and the clearest risk signal is the **assist-to-execute line** between drafting something for a person and executing a change in a system of record.

Classify each candidate action along three axes:

- **Operational risk**: what breaks if the action is wrong? Consider blast radius (one file, one service, every consumer), reversibility (a revert commit versus a dropped table or a published package) and detectability (will a failing check

catch it, or will it surface in production).

- **Security risk**: does the action touch secrets, credentials, privilege, network egress, the software supply chain (new dependencies, workflow files, actions versions) or the agent's own configuration (instructions, custom agents, MCP server configuration, hooks)?
- **Compliance risk**: does the action affect regulated data, segregation of duties, change-management evidence or audit records? An agent that can approve or merge its own work, or delete evidence such as workflow artifacts, has a compliance problem even when the code is correct.

Responsible AI considerations sit across all three axes: fairness and privacy (does the change process personal data differently?), transparency (can a reviewer see what the agent did and why?) and accountability (is there a named human who owns the outcome?).

Table 6.1: A working risk matrix for agent actions in a GitHub SDLC

Action class	Operational	Security	Compliance	Overall
Read code, search issues, summarise a failing log, draft a plan	None	Low (data exposure only)	None	**Low**
Edit application code or tests on an agent branch, open a draft PR	Low (reversible, reviewed)	Low to medium	Low	**Low to medium**
Add or upgrade a dependency, change build scripts	Medium	Medium to high (supply chain)	Medium (licence)	**Medium to high**
Edit `.github/workflows/*`, `copilot-setup-steps.yml`, hooks, custom agents, MCP configuration, CODEOWNERS	Medium	**High** (privilege escalation path)	High	**High**
Infrastructure as code under `/infra`, IAM, network rules	High	High	High	**High**
Deploy to production, run a data migration, access production secrets	High, often irreversible	High	High	**Critical**
`rm -rf` on shared paths, force-push, `sudo`, piping remote scripts to a shell, deleting audit evidence	Critical	Critical	Critical	**Prohibited**

EXAM TIP

When a stem lists several agent actions and asks which require human involvement, pick the actions that are **irreversible**, that **approve or accept** the agent's own work, or that change **security-relevant configuration**. Analysis tasks (generating an assessment, checking that a file exists, ordering dependencies) are safe to delegate; **accepting** that analysis and approving commits are not.

6.1.2 Right-sizing human intervention

The output of classification is a decision about **where** a human sits relative to the action:

- **Human out of the loop**: the action runs without a person, but within hard limits (deny rules, firewall, least-privilege token). Appropriate only for low-risk, reversible actions.
- **Human on the loop**: the agent acts, automation verifies (required status checks, code scanning, secret scanning), and a person reviews the result before it becomes permanent (the draft PR review and merge).
- **Human in the loop**: the action cannot start until a named person authorises it (environment required reviewers, "Approve and run workflows").
- **Human only**: the agent is technically prevented from doing it at all (a `preToolUse` deny, a `--deny-tool` rule, a ruleset the agent cannot bypass).

Microsoft's tiered governance model maps neatly onto this: low-risk (Tier 1) agents get a named owner, basic monitoring and self-service deployment within published guardrails; medium-risk (Tier 2) agents add a domain-expert validator and a formal release gate; high-risk (Tier 3) agents add a security review and Responsible AI assessment before release, a **decision-rights framework** that "sets what the agent may decide alone and what needs a human", and an incident-response plan. Agents start in the tier that fits their behaviour and move up as scope or autonomy grows.

6.1.3 Assigning autonomy levels

GitHub does not ship a setting called "autonomy level"; the levels below are a study model. What the exam tests is that you can map a risk class to the **right combination of existing controls**, without adding controls that slow delivery but do not reduce risk.

Table 6.2: Decision table, risk level to autonomy level to enforcing control

Risk level	Autonomy level	Human role	Enforcing GitHub controls
Low	**A3 Autonomous**: act and verify automatically	Out of / on the loop	Required status checks; built-in CodeQL, secret scanning and dependency checks of the cloud agent; CLI `--allow-tool` for known-safe commands; session-scoped approvals in VS Code
Medium	**A2 Act, human accepts**: agent works on its own branch, a human merges	On the loop	Draft PR plus ruleset "Require a pull request before merging" with required approvals; agent cannot approve or merge; requester cannot approve their own delegated PR
High	**A1 Act after authorisation**: designated people must approve specific paths or runs	In the loop	CODEOWNERS plus "Require review from Code Owners"; "Approve and run workflows" left enabled for agent PRs; ruleset required reviewers for sensitive paths
Critical	**A0 Propose only**: agent prepares, human executes or approves each run	In the loop, named approver	Environment with required reviewers, prevent self-reviews, deployment branch restrictions, admin bypass disabled
Prohibited	**Blocked**	Human only	`preToolUse` hook returning `deny`; CLI `--deny-tool`; firewall; push protection; Agents secrets scoping (secret never reaches the agent)

COMMON TRAP

"Read-only analysis" and "autonomous in non-production only" sound safe, but if the scenario requires the agent to **actually update and deploy** something to production, they fail the requirement. The correct answer for a compliance-sensitive production change is usually **human-in-the-loop with an explicit approval gate** on the production step, not removing the agent's ability to do the work.

6.1.4 Defaults you build on: the cloud agent's built-in boundaries

Before adding controls, know what Copilot cloud agent already enforces. GitHub documents these mitigations in *Risks and mitigations for GitHub Copilot cloud agent*:

Risk	Built-in mitigation
Untrusted people steering the agent	Only users with write access can trigger the agent; comments from users without write access are never presented to it.
Unreviewed pushes	The agent can push only to a single branch (a new `copilot/` branch, or the PR branch when invoked with `@copilot` on an existing PR). It cannot run `git push` directly and is subject to branch protections and required checks.
Self-approval	Draft PRs must be reviewed and merged by a human; the agent cannot mark a PR ready for review, approve or merge. The user who asked Copilot to create the PR cannot approve it.

Risk	Built-in mitigation
Unattributed PRs	When the agent opens a PR under its own app identity, one more approval is required if the repository already requires at least one (on by default in rulesets, always on for branch protection). PREVIEW
Privileged workflows on agent code	By default, workflows do not run on Copilot's pushes until a user with write access clicks **Approve and run workflows**.
Data exfiltration	Internet access is restricted by the agent firewall.
Prompt injection	Hidden characters, such as HTML comments in issues and PR comments, are filtered before input reaches the agent.
Vulnerable output	CodeQL, secret scanning, GitHub Advisory Database checks on new dependencies and Copilot code review run by default, without requiring a GitHub Advanced Security licence.
Loss of oversight	Commits are authored by Copilot with the requesting developer as co-author, are signed (shown as Verified), and link to the session log; session logs and audit log events are available to administrators.

For automations (event or schedule driven cloud agent runs) GitHub adds that work is attributed to the person who created the automation, you choose exactly which tools the automation can use, events from users without write access are ignored by default, and workflows still require human approval, which stops automation chains from running workflows unattended.

6.2 Implement guardrails and human-in-the-loop workflows

6.2.1 Identify the subset of actions that require human judgment

Human judgment is a scarce resource. Spend it where a person adds information the agent and the automation do not have. A practical test has four questions; if any answer is "yes", the action needs a human decision point:

1. **Is it irreversible or expensive to reverse?** Production deployments, data migrations, deleting branches or artifacts, publishing packages, rotating credentials.
2. **Does it accept or approve work?** Approving PRs, approving commits, marking an assessment or plan as correct, approving a workflow run. An agent must never be the final reviewer of its own output.
3. **Does it change the guardrails themselves?** Workflow files, `copilot-setup-steps.yml`, `.github/hooks/*.json`, custom agents in `.github/agents/`, `.github/copilot-instructions.md`, `AGENTS.md`, MCP configuration, CODEOWNERS, rulesets.
4. **Does it need context outside the repository?** Business priority, legal or licensing judgement, customer impact, a security exception.

Everything else (writing code on a branch, running tests, generating a plan, summarising a failure) can be delegated and verified after the fact by checks and review. GitHub's guardrails tutorial explicitly recommends protecting "important Copilot and MCP configuration files with a `CODEOWNERS` file" and enabling "Require review from Code Owners", so that edits to agent configuration need approval from specific teams.

> **NOTE**
> Instructions files are guidance, not enforcement. Writing "never edit workflow files" in `.github/copilot-instructions.md` or `AGENTS.md` reduces the chance the agent tries; only a hook, deny rule, ruleset, CODEOWNERS requirement or environment protection rule actually prevents the outcome.

6.2.2 Block policy violations with hooks

Hooks are external commands (or HTTP calls) that run at lifecycle points during an agent session. They are supported in **Copilot CLI** and **Copilot cloud agent**. In a cloud agent job the only configuration loaded is `.github/hooks/*.json` from the cloned repository; the CLI additionally loads user-level files (`~/.copilot/hooks/`), inline `hooks` blocks in settings, plugin hooks and machine-wide **policy hooks**.

Table 6.3: Hook events most relevant to guardrails

Event	Fires when	Can it prevent the action?	Cloud agent behaviour
`preToolUse`	Before each tool executes	**Yes**: allow, deny or modify arguments	Fires; `"ask"` is treated as `"deny"`
`permissionRequest`	Before the CLI permission service runs	Yes (CLI only)	No effect: tools are pre-approved; use `preToolUse`
`postToolUse`	After a tool completes successfully	No: can only modify the result or add context	Fires
`postToolUseFailure`	After a tool completes with a failure	No: recovery guidance only	Fires
`errorOccurred`	An error occurs during execution	No (output not processed)	Fires; good for diagnostics and escalation
`sessionStart` / `sessionEnd`	Session begins / terminates	No (`sessionStart` can add context)	Once per job; end `reason` typically `complete`, `error` or `timeout`
`userPromptSubmitted`	The user submits a prompt	No for config-file hooks (output dropped)	At most once per job
`agentStop` / `subagentStop`	Agent or subagent finishes a turn	Can `block` to force another turn	Fires; extra turns count against the job timeout

Key `preToolUse` semantics to memorise:

- The hook writes JSON to stdout with `permissionDecision` (`"allow"`, `"deny"` or `"ask"`), `permissionDecisionReason` (shown to the agent and **required** when denying) and optionally `modifiedArgs`. Empty output means default behaviour.
- If several hooks run for the same event and **any** returns `"deny"`, the tool is blocked.
- Command hooks are **fail-closed**: a crash, exit code `2` or any other non-zero exit denies the tool call, even if stdout says `"allow"`.
- **Timeouts are fail-open** for every event, including `preToolUse` and admin policy hooks: the tool proceeds through the normal permission flow. Default `timeoutSec` is 30.
- HTTP `preToolUse` hooks are fail-open on network errors, timeouts or non-2xx responses, and must use `https://`.
- An optional `matcher` regex (anchored as `^(?:PATTERN)$`) filters by tool name, for example `bash` or `bash|edit`.

Listing 6.1: `.github/hooks/guardrails.json` with a pre-tool deny gate, failure escalation and an audit sink

```
{
  "version": 1,
  "hooks": {
    "preToolUse": [
      {
        "type": "command",
        "matcher": "bash",
        "bash": "./.github/hooks/scripts/deny-dangerous.sh",
        "timeoutSec": 10
      }
    ],
    "postToolUse": [
```

```
      {
        "type": "http",
        "url": "https://hooks.example.com/copilot-audit",
        "timeoutSec": 10
      }
    ],
    "errorOccurred": [
      {
        "type": "command",
        "bash": "./.github/hooks/scripts/on-error.sh",
        "timeoutSec": 15
      }
    ]
  }
}
```

Listing 6.2: `deny-dangerous.sh`, a `preToolUse` script that denies destructive shell commands

```
#!/usr/bin/env bash
# Reads the preToolUse payload on stdin: {sessionId, timestamp, cwd, toolName, toolArgs}
set -euo pipefail

INPUT="$(cat)"
TOOL="$(printf '%s' "$INPUT" | jq -r '.toolName')"
[ "$TOOL" = "bash" ] || exit 0          # empty output: default permission flow

# toolArgs may arrive as a JSON string or an object
CMD="$(printf '%s' "$INPUT" | jq -r '.toolArgs
  | (if type == "string" then fromjson else . end)
  | .command // ""')"

deny() {
  jq -cn --arg r "$1" '{permissionDecision: "deny", permissionDecisionReason: $r}'
  exit 0
}

if printf '%s' "$CMD" | grep -Eq '(^|[;&|[:space:]])sudo[[:space:]]'; then
  deny "Privilege escalation (sudo) is blocked by repository policy"
fi
if printf '%s' "$CMD" | grep -Eq 'rm[[:space:]]+-[a-zA-Z]*r[a-zA-Z]*f?[[:space:]]+/'; then
  deny "Recursive delete of an absolute path is blocked: destructive_operation_detected"
fi
if printf '%s' "$CMD" | grep -Eq '(curl|wget)[^|]*\|[[:space:]]*(ba|z)?sh'; then
  deny "Piping remote content into a shell is blocked by repository policy"
fi
if printf '%s' "$CMD" | grep -Eq 'git[[:space:]]+push.*(--force|-f)'; then
  deny "Force-push is blocked; open a pull request instead"
fi
exit 0
```

PRODUCTION NOTE

Pattern matching on command strings is defence in depth, not a sandbox. Aliases, variable expansion, symlinks and indirect writes can evade regular expressions. Keep the script fast (a timeout means fail-open), emit exactly one final JSON object (two objects concatenate into invalid JSON and are ignored), and remember that if `jq` is missing the script crashes and, because command hooks are fail-closed, every `bash` call is denied. Pair hooks with OS-level isolation, the firewall and least-privilege credentials.

Other hook details that appear in scenarios:

- The cloud agent sandbox is Linux; only `bash` (or the cross-platform `command`) fields are honoured, the working directory is `/workspace`, the filesystem is ephemeral, and `GITHUB_TOKEN` is not set. To keep hook output, send it to an `http` hook, whose host must be allowed by the firewall.

- **Policy hooks** (CLI only) live in `/etc/github-copilot/policy.d/*.json` on Linux and macOS or `C:\ProgramData\GitHub\Copilot\policy.d\*.json` on Windows, must be root-owned and not group- or world-writable on POSIX, load before all other hooks, and cannot be disabled by `disableAllHooks`. They are the right answer when IT must enforce a rule developers cannot switch off on their own machines.
- `disableAllHooks: true` inside a single `.github/hooks/*.json` file skips only that file's hooks (honoured by CLI and cloud agent).
- In VS Code, a `PreToolUse` hook returning `permissionDecision: "deny"` is the documented way to **block** a terminal tool call in the Local harness PREVIEW; a `false` terminal auto-approve rule only requires approval.

6.2.3 Block policy violations with tool permissions (CLI and VS Code)

Copilot CLI controls tools with permission patterns of the form `Kind(argument)`, where the kind is `shell`, `write`, `read`, `url`, `memory` or an MCP server name. The rules that matter:

- `--allow-tool` grants permission without prompting; `--deny-tool` refuses without prompting.
- **Deny rules always take precedence over allow rules, even when `--allow-all` is set.**
- `shell(git:*)` matches the command stem followed by a space, so it matches `git push` but not `gitea`.
- `--deny-tool='write(PATH)'` scopes a denial to one path; the match resolves symlinks and `.`/`..` segments.
- `--available-tools` and `--excluded-tools` change which tools the model can **see**; allow and deny rules control which calls are **permitted**.
- `--allow-all-tools` (env `COPILOT_ALLOW_ALL`) is required for programmatic use. Administrators can suppress all allow-all options (`--allow-all-tools`, `--allow-all-paths`, `--allow-all-urls`, `--allow-all`, `--yolo`) by setting `permissions.disableBypassPermissionsMode` to `"disable"`, from user settings, managed settings or an MDM policy.

Listing 6.3: Copilot CLI allow and deny rules for an unattended run

```
# Interactive: allow any git command except push, never touch .env
copilot --allow-tool='shell(git:*)' \
        --deny-tool='shell(git push)' \
        --deny-tool='write(.env)'

# Programmatic run in CI: pre-approve tools, but deny the dangerous ones.
# Deny wins over --allow-all-tools.
copilot -p "Fix the failing unit tests in src/billing and summarise the change" \
  --allow-all-tools \
  --deny-tool='shell(rm:*),shell(sudo:*),shell(git push)'

# Restrict MCP usage to a single tool from one server
copilot --allow-tool='MyMCP(create_issue)'
```

In VS Code agent mode, the equivalent levers are the tools picker, approval scopes (session, workspace, user), `chat.tools.terminal.autoApprove` rules (a `false` rule always takes precedence and forces approval), `chat.tools.edits.autoApprove` with patterns such as `"**/.env": false` to force approval of sensitive file edits, `chat.tools.eligibleForAutoApproval` to make a tool always require manual approval, and agent sandboxing. VS Code states plainly that terminal auto-approval is "a best-effort convenience, not a security boundary" and that the Allow all and Autopilot permission levels bypass approvals.

COMMON TRAP

A `postToolUse` hook, a `sessionStart` banner or a `userPromptSubmitted` logger cannot stop a command. If the requirement is "prevent the command from running", the answer is a `preToolUse` deny (cloud agent or CLI) or a `--deny-tool` rule (CLI).

6.2.4 Block data leakage: content exclusion, firewall and push protection

Content exclusion tells Copilot to ignore files. With Copilot Business or Enterprise, repository administrators, organisation owners and enterprise owners configure it under **Settings > Copilot > Content exclusion**. Excluded files lose inline suggestions, do not inform suggestions or responses, and are not reviewed by Copilot code review. Patterns use fnmatch notation and are case insensitive. Changes can take up to 30 minutes to reach IDEs that have already loaded settings. Changes are audited as `copilot.content_exclusion_changed`.

Listing 6.4: Content exclusion at repository level (first block) and organisation level (second block)

```
# Repository settings: Paths to exclude in this repository
- "/deploy/templates/credentials/**"
- "secrets.json"
- "*.pem"

# Organisation settings: Repositories and paths to exclude
"*":
  - "**/.env"
payments-service:
  - "/config/prod/**"
https://github.com/octo-org/billing.git:
  - "/fixtures/customer-data/**"
```

> **COMMON TRAP**
> Content exclusion is **not** a universal agent boundary. GitHub documents that it is not supported in the Edit and Agent modes of Copilot Chat in VS Code and other editors, and the how-to page states that Copilot CLI and agent mode in IDEs do not support it. It also does not apply to symlinks or remote filesystems, and IDEs may still pass semantic information (types, hover definitions) indirectly. If an agent must never read a file, keep the file out of the agent's environment and credentials out of its reach. There is no documented `.copilotignore` file.

The **Copilot cloud agent firewall** limits internet access to reduce exfiltration risk. GitHub and Copilot hosts are always allowed; a **recommended allowlist** (on by default) lets the agent download dependencies; organisation owners can manage an organisation custom allowlist, decide whether repositories may add custom rules, and set the recommended allowlist to Enabled, Disabled or "Let repositories decide". A blocked request adds a warning to the PR body or a comment showing the blocked address and the command. Its documented limitations are exam material: it applies only to processes the agent starts through its Bash tool, **not** to MCP server processes or Copilot setup steps, only within the GitHub Actions appliance, and sophisticated attacks may bypass it.

Push protection blocks secrets before they reach the repository, in pushes from the command line, commits in the UI, file uploads, REST API requests and GitHub MCP server interactions (public repositories only). Repository push protection requires GitHub Secret Protection and is disabled by default; bypasses generate alerts and the `secret_scanning_push_protection.bypass` audit event. Independently, the cloud agent runs secret scanning on its own output without needing that licence.

6.2.5 Scope permissions and execution contexts for least privilege

Least privilege for agents has four layers: the **token** the work runs with, the **secrets** the agent can see, the **runner** it executes on and the **scope** of repositories and branches it can touch.

Workflow token. In any workflow (including one that invokes Copilot CLI or another agent), the `permissions` key sets `GITHUB_TOKEN` scopes at workflow or job level. If you specify access for any permission, every unspecified permission becomes `none`; `read-all`, `write-all` and `{}` are shorthand forms. A job-level block replaces the workflow-level block for that job. The restricted repository default gives read access to `contents` and `packages` only, and workflows cannot create or approve pull requests unless **Allow GitHub Actions to create and approve pull requests** is enabled. Note that `pull_request_target` receives a read/write token even from forks.

Listing 6.5: Read-only default with job-level write access only where an agent writes

```yaml
name: agent-maintenance
on:
  pull_request:
  workflow_dispatch:

permissions: read-all          # every job starts read-only

jobs:
  analyse:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: ./scripts/agent-analyse.sh   # reads code, writes a report artifact

  regenerate:
    if: github.event_name == 'workflow_dispatch'
    runs-on: ubuntu-latest
    permissions:                # replaces read-all for this job only
      contents: write           # push the regenerated files to a branch
      pull-requests: write      # open the pull request
    steps:
      - uses: actions/checkout@v4
      - run: ./scripts/agent-regenerate.sh
```

EXAM TIP

If the stem says "default read-only, write only for the job that pushes and opens a PR", the job needs `contents: write` and `pull-requests: write`; `issues: write` is not needed to create a pull request. Placing write at workflow level is wrong because it also grants write to the analysis job. Use `copilot-requests: write` only when a workflow uses Copilot CLI with `GITHUB_TOKEN`.

Copilot cloud agent token and setup steps. The enterprise `GITHUB_TOKEN` default permissions policy does **not** affect the token Copilot receives for its session, but it does apply to `copilot-setup-steps.yml`. The job must be called `copilot-setup-steps`, the file must be on the default branch, `timeout-minutes` can be at most 59, and GitHub's template comments ask you to set permissions to the lowest needed (usually `contents: read` for checkout).

Listing 6.6: Least-privilege `.github/workflows/copilot-setup-steps.yml`

```yaml
name: "Copilot Setup Steps"
on:
  workflow_dispatch:
  push:
    paths: [.github/workflows/copilot-setup-steps.yml]
  pull_request:
    paths: [.github/workflows/copilot-setup-steps.yml]

jobs:
  copilot-setup-steps:          # the job name must be exactly this
    runs-on: ubuntu-latest
    timeout-minutes: 30
    permissions:
      contents: read            # enough to clone and install dependencies
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: "20"
      - run: npm ci
```

Secrets. Copilot cloud agent sees only **Agents** secrets and variables (repository or organisation level); it has no access to Actions, Codespaces or Dependabot secrets. Names prefixed `COPILOT_MCP_` are available only to MCP servers. GitHub's guidance is therefore simple: keep anything the agent must not touch as Actions secrets, and give the agent the narrowest Agents secret it needs. Production deployment credentials belong in an **environment** secret protected by required reviewers (6.2.6).

Runners. GitHub recommends GitHub-hosted runners so each session runs on a fresh VM; if you use self-hosted runners, make them ephemeral. Organisation owners can pin the agent to a runner label for all repositories.

Scope. The cloud agent works in one repository per task, on one branch, and opens one PR per task; the GitHub MCP server by default sees only that repository. Enterprise and organisation policies decide which organisations and repositories can use the agent and which MCP servers it may call. Treat every extra MCP tool and every allowlisted domain as additional privilege.

6.2.6 Controlled paths for irreversible or compliance-sensitive changes

Some changes must pass through a gate that a named human controls. GitHub offers four complementary gates; know which moment each one guards.

Table 6.4: Which gate guards which moment

Requirement	Control	Moment it acts
Workflows must not run on agent pushes until someone has looked at the code	**Require approval for workflow runs** (Copilot cloud agent setting, on by default); approver clicks **Approve and run workflows** in the PR merge box	Before any workflow runs on the agent's branch
Specific paths (workflows, infra, agent configuration) need approval from designated owners	**CODEOWNERS** plus **Require review from Code Owners** (ruleset or branch protection)	Before merge
All changes to `main` via PR with N approvals, checks green, no force-push	**Ruleset** (or branch protection) on the default branch	Before merge / on push
A deployment job must wait for a named person	**Environment** with required reviewers, referenced by `jobs.<job_id>.environment`	Before the job is sent to a runner

CODEOWNERS. Place the file in `.github/`, the repository root or `docs/` (searched in that order). Patterns follow most gitignore rules, the **last matching pattern wins**, paths are case sensitive, owners (users or teams) must have write permission, and the file must be on the PR's base branch and under 3 MB. On its own, CODEOWNERS only requests reviews; enforcement comes from "Require review from Code Owners", after which approval from **any** one owner of the changed file satisfies the rule.

Listing 6.7: `.github/CODEOWNERS` protecting agent configuration, workflows and infrastructure

```
# Default owners for everything
*                                   @octo-org/app-maintainers

# Agent configuration and guardrails
/.github/copilot-instructions.md    @octo-org/platform-security
/.github/instructions/              @octo-org/platform-security
/.github/agents/                    @octo-org/platform-security
/.github/hooks/                     @octo-org/platform-security
/AGENTS.md                          @octo-org/platform-security

# Workflows, including copilot-setup-steps.yml
/.github/workflows/                 @octo-org/platform-security @octo-org/release-eng

# Infrastructure as code
/infra/                             @octo-org/cloud-platform
```

```
# Protect the ownership map itself
/.github/CODEOWNERS                @octo-org/platform-security
```

Rulesets. GitHub's guardrails tutorial notes that the cloud agent is already restricted from pushing to the default branch or merging, and "is subject to rulesets just like human developers". Useful rules: **Require a pull request before merging** (with required approvals, code owner review, dismiss stale approvals, approval of the most recent reviewable push, conversation resolution), **Require status checks to pass**, **Require code scanning results**, **Require deployments to succeed**, **Block force pushes**, **Restrict deletions** and push rules such as **Restrict file paths**. A ruleset that only allows specific commit authors can block the agent; with rulesets, administrators can add Copilot as a bypass actor. Copilot can sign commits, so "Require signed commits" is compatible.

Listing 6.8: Branch ruleset for the default branch (REST API body for `POST /repos/OWNER/REPO/rulesets`)

```
{
  "name": "main-agent-guardrails",
  "target": "branch",
  "enforcement": "active",
  "conditions": {
    "ref_name": { "include": ["~DEFAULT_BRANCH"], "exclude": [] }
  },
  "rules": [
    { "type": "deletion" },
    { "type": "non_fast_forward" },
    {
      "type": "pull_request",
      "parameters": {
        "required_approving_review_count": 1,
        "require_code_owner_review": true,
        "dismiss_stale_reviews_on_push": true,
        "require_last_push_approval": true,
        "required_review_thread_resolution": true
      }
    },
    {
      "type": "required_status_checks",
      "parameters": {
        "strict_required_status_checks_policy": true,
        "required_status_checks": [ { "context": "build-and-test" } ]
      }
    }
  ]
}
```

> **NOTE**
> With rulesets, **Require an additional approval for unattributed Copilot pull requests** is enabled by default PREVIEW. If the ruleset requires one approval and Copilot opens a PR under its own app identity (for example from a shared Slack or Teams thread), two approvals from people with write access are needed. The setting has no effect when zero approvals are required. Rulesets also support **required reviewers** per file pattern (up to 15 teams), which is in beta in the REST API.

Environments with required reviewers. An environment's deployment protection rules must all pass before a job that references the environment is sent to a runner. **Required reviewers** lists up to six users or teams (each needing at least read access to the repository); only **one** of them must approve. **Prevent self-reviews** stops the person who initiated the deployment from approving it. Other rules: a **wait timer** (1 to 43,200 minutes), **deployment branches and tags** (no restriction, protected branches only, or selected patterns), disallowing **administrator bypass** (bypass is allowed by default), and custom protection rules powered by GitHub Apps PREVIEW, with at most six rules enabled per environment. On Free, Pro and Team plans, required reviewers and wait timers are available only for public repositories.

Listing 6.9: Deployment workflow where only the production job waits for a human

```yaml
name: deploy
on:
  push:
    branches: [main]

permissions:
  contents: read

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: ./scripts/build.sh

  deploy-staging:
    needs: build
    runs-on: ubuntu-latest
    environment: staging              # no reviewers: deploys automatically
    steps:
      - run: ./scripts/deploy.sh staging

  deploy-production:
    needs: deploy-staging
    runs-on: ubuntu-latest
    environment:
      name: production                # required reviewers + prevent self-reviews
      url: https://app.example.com
    permissions:
      contents: read
      id-token: write                 # OIDC to the cloud, no long-lived secret
    steps:
      - run: ./scripts/deploy.sh production
```

COMMON TRAP

A **concurrency group** limits parallelism, not authorisation. A **custom deployment protection rule** powered by a GitHub App automates the decision through a third-party system; if the requirement is that a **person** explicitly approves, choose required reviewers. "Approve and run workflows" gates **whether workflows run** on an agent PR; it does not gate a later production deployment, which needs the environment.

6.2.7 Workflow approval for agent pull requests

By default, GitHub Actions workflows do not run automatically when Copilot pushes to a PR. GitHub explains why: workflows "can be privileged and have access to sensitive secrets", so a reviewer should inspect the changes, paying particular attention to `.github/workflows/`, and then click **Approve and run workflows**. A repository administrator can disable **Require approval for workflow runs** under **Settings > Copilot > Cloud agent > Actions workflow approval**, but GitHub warns that this "may allow unreviewed code written by Copilot to gain write access to your repository or access your GitHub Actions secrets". The guardrails tutorial asks enterprise owners to agree the preferred setting with repository administrators in advance.

Validation workflows should use `on: pull_request` so they evaluate the PR itself and can be made required status checks. Branch protection on `main` does not stop the agent from working, because it only pushes to its own branch and opens a PR. Do not add bypasses for direct pushes to "help" the agent; that defeats the PR-only requirement.

6.2.8 Preserve velocity by minimising approvals that do not reduce risk

Every approval has a cost: it blocks the agent, interrupts a person and, when overused, trains reviewers to click through. Microsoft's governance guidance says to "keep the path light" for low-risk work and to reserve heavier controls for agents that can cause real harm. Practical rules:

- **Replace human checks with automated ones where the human adds nothing.** A required status check, CodeQL, secret scanning and the agent's built-in validation tools are faster and more consistent than a person reading test logs.
- **Gate the irreversible step, not every step.** Let the agent build and deploy to staging freely; put required reviewers only on `production` (Listing 6.9).
- **Scope reviews by path.** CODEOWNERS and ruleset required reviewers pull in the security team only when workflows, infrastructure or agent configuration change, not on every PR.
- **Block instead of asking.** For actions that are never acceptable, a `preToolUse` deny or `--deny-tool` is cheaper than a prompt someone has to answer. Under the cloud agent there is nobody to ask anyway: `"ask"` becomes `"deny"`.
- **Pre-approve the known safe set.** CLI `--allow-tool='shell(npm test)'` or VS Code session-level approvals remove prompts for routine commands while deny rules still win.
- **Do not stack duplicate approvals.** Two required approvals plus code owner review plus an environment approval on a documentation change adds delay without reducing risk.
- **Measure.** If an approval is granted almost every time with no changes requested, it is a candidate for automation or removal; if it regularly catches problems, keep it.

EXAM TIP

When two answers are both "safe", prefer the one that keeps the human gate only on the high-risk step and uses automation elsewhere. The exam rewards right-sized control, not maximum control.

6.3 Accountability: attribution, audit and retention

6.3.1 Attribute actions to the agent and to the human who delegated

Accountability needs two answers for every change: **which agent acted** and **which person is responsible**. GitHub records both.

- **Commits**: authored by Copilot, with the developer who assigned the issue or requested the change as co-author; signed and shown as Verified; the commit message links to the agent session log.
- **Session logs**: show the analysis performed and actions taken, including security validation results.
- **Audit log events for agents**: filter with `actor:Copilot` to see agentic activity over the last 180 days. Key fields:

Table 6.5: Agentic audit log fields

Field	Meaning	Example
`action`	The action the agent performed	`pull_request.create`
`actor_is_agent`	Whether the actor is an AI agent; always `true` for agentic events	`true`
`agent_session_id`	Links the event to the specific agent session; present only when the event results from an agent session	a UUID
`user`	The person who initiated the agentic event (the user for whom the agent acted)	`octocat`

Many standard events carry these fields; for example `pull_request.create` lists `actor_is_agent` and `agent_session_id` among its additional fields, and events such as `workflows.approve_workflow_job`, `environment.add_protection_rule` and `copilot.content_exclusion_changed` include `actor_is_agent`. Enterprise owners can also see active and recent agent sessions under **AI controls** (the three most recent, with **View all** for the last 24 hours) and filter them.

COMMON TRAP

The `user` field tells you **who delegated**, not that an agent acted. To prove agent initiation, use `actor_is_agent` and `agent_session_id` (or the `actor:Copilot` filter). The audit log does not contain local prompts from a developer's machine.

6.3.2 Search the audit log

The enterprise audit log lists events from the last 180 days (Git events for seven days), shows only the past three months by default (use `created` for older events), and cannot be searched by free text: queries are built from `key:value` qualifiers. Event names are `category.operation`, for example `repo.create`. Git events are not included in UI search results; use the API with `include=git`.

Table 6.6: Audit log qualifiers

Qualifier	Use	Example
`actor`	Account (person or agent) that performed the action; prefix `-` to exclude	`actor:Copilot`, `-actor:Copilot`
`user`	User affected; for agent actions, the user the agent acted for	`user:octocat`
`action`	Specific event or whole category	`action:artifact.destroy`, `action:team`
`operation`	Generic type: `create`, `access`, `modify`, `remove`, `authentication`, `transfer`, `restore`	`operation:remove`
`repo`	Repository, always with the owner	`repo:octo-org/payments`
`created`	Date or range	`created:2026-09-01..2026-09-30`
`country`, `ip`, `hashed_token`	Origin and token-based investigations	`country:de`

Listing 6.10: Audit log queries for agent oversight (UI search box and REST API)

```
# PRs opened by Copilot in the last 30 days, across the enterprise
actor:Copilot action:pull_request.create created:>=2026-09-19

# Everything agents did on behalf of one developer in one repository
actor:Copilot user:octocat repo:octo-org/payments

# Who manually deleted workflow run artifacts (review evidence) in a repo
action:artifact.destroy repo:octo-org/payments

# Who approved jobs waiting on an environment, and who changed its protection rules
action:workflows.approve_workflow_job repo:octo-org/payments
action:environment.update_protection_rule

# Changes to guardrails themselves
action:repository_ruleset.update
action:protected_branch.update_require_code_owner_review
action:copilot.content_exclusion_changed

# Same search through the REST API (token needs the read:audit_log scope)
curl -H "Authorization: Bearer $TOKEN" \
  "https://api.github.com/enterprises/octo-ent/audit-log?
phrase=actor:Copilot+action:pull_request.create+created:>=2026-09-19&per_page=100"
```

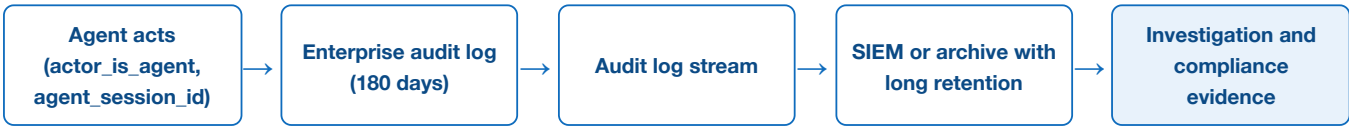
EXAM TIP

Pick the **specific event** over a generic qualifier. "Who deleted the artifacts?" is `action:artifact.destroy` (a workflow run artifact was manually deleted), not `operation:remove` (too broad), `action:workflows` (runs, not artifacts) or `checks.delete_logs` (check suite logs). Combine with `repo:` to narrow. Artifact deletion is an Actions event, not a Copilot event.

6.3.3 Retain evidence beyond the default window: audit log streaming

The built-in 180-day window cannot be extended. When policy requires longer retention (for example two or seven years), **stream** the audit log to an external system that you control. Streaming exports audit and Git events for all organisations in the enterprise from the moment the stream is enabled, as compressed JSON files named `YYYY/MM/DD/HH/MM/<uuid>.json.log.gz`. Supported destinations are Amazon S3 (access keys or OIDC), Azure Blob Storage, Azure Event Hubs, Datadog, Google Cloud Storage, Splunk and, for Copilot agent session events only, Microsoft Purview. Streaming to multiple endpoints is in public preview.

- **Retention** is whatever the destination is configured for; streaming itself only transports the events. If a stream feeds Event Hubs, the two-year archive must exist downstream.
- **Continuity**: a paused stream keeps a seven-day buffer. Paused for more than seven days, it resumes from one week before the current time; paused for three weeks or more, it starts anew with no retained data.
- **Health checks** run every 24 hours; enterprise owners are emailed about misconfiguration, which must be fixed within six days to avoid dropped events.
- **Copilot agent session events** PREVIEW: for enterprises using Enterprise Managed Users or GHE.com data residency, enable **Copilot Usage Records Streaming** ("Enabled everywhere") under **AI Controls** and configure a destination. Each streamed usage record includes `type` (`request` or `response`), `user_id`, `enterprise_id`, `endpoint`, `body`, `@timestamp`, `truncated`, `event_id` and `github_request_id`.



PRODUCTION NOTE

Protect the evidence trail itself. Restrict who can delete workflow artifacts and logs, alert on `artifact.destroy` and `checks.delete_logs`, and configure access control, ingestion monitoring and de-duplication in the downstream store. Hook output written inside a cloud agent sandbox is discarded when the job ends; send anything you need to keep to an `http` hook endpoint (allowed through the firewall) or rely on the audit stream.

6.4 Key terms

Autonomy level	The degree of independent action granted to an agent for a class of actions, from blocked through propose-only and act-after-authorisation to fully autonomous. Assigned per action based on risk.
Human-in-the-loop (HITL)	A design in which a named person must authorise an action before it executes, for example an environment required reviewer.
Decision-rights framework	A Tier 3 governance control that defines what an agent may decide alone and what needs a human.
Hook	An external command or HTTP call that runs at a lifecycle event (`preToolUse`, `postToolUse`, `errorOccurred`, `sessionStart`, `sessionEnd` and others) in Copilot CLI or Copilot cloud agent, configured in `.github/hooks/*.json` with `"version": 1`.
Fail-closed / fail-open	Whether a failing guard blocks or allows. Command `preToolUse` hooks fail closed on errors and non-zero exits; all hook timeouts and HTTP `preToolUse` failures fail open.

Policy hook	A machine-wide CLI hook installed by administrators under a root-owned policy directory; loads first and cannot be disabled with `disableAllHooks`.
Content exclusion	Copilot Business/Enterprise setting that stops Copilot using specified files on supported surfaces; not a boundary for IDE agent mode or the CLI.
Agent firewall	Network allowlisting for Copilot cloud agent processes started through its Bash tool; does not cover MCP server processes or setup steps.
Agents secrets and variables	The only secrets and variables passed to Copilot cloud agent; `COPILOT_MCP_`-prefixed ones go only to MCP servers.
Approve and run workflows	Merge-box button a user with write access clicks to let workflows run on Copilot's pushes; required by default.
Required reviewers (environment)	Up to six users or teams, one of whom must approve a job that references the environment before it reaches a runner.
Unattributed Copilot pull request	A PR Copilot opens under its own app identity rather than on behalf of a person; rulesets require one extra approval by default.
`actor_is_agent` / `agent_session_id`	Audit log fields that mark an event as agent-initiated and link it to a specific agent session.
Audit log streaming	Continuous export of enterprise audit and Git events to an external store (S3, Azure Blob Storage, Event Hubs, Datadog, Google Cloud Storage, Splunk) for retention and analysis beyond 180 days.

6.5 Additional practice questions

Practice 6.1 MULTIPLE CHOICE

Your security team adds a `preToolUse` command hook to `.github/hooks/scan.json`. The hook calls an internal malware scanner over the network and sometimes takes longer than its `timeoutSec` of 15 seconds. You need to know how Copilot cloud agent treats a shell command when the hook times out. What happens?

A. The command is denied because `preToolUse` command hooks are fail-closed.
B. The command proceeds through the normal permission flow because hook timeouts are fail-open.
C. The agent pauses and asks the user who assigned the issue for approval.
D. The whole session ends with `reason: "timeout"`.

Practice 6.2 MULTIPLE CHOICE

A nightly job runs Copilot CLI non-interactively with `--allow-all-tools` so it never stalls on prompts. The agent must be able to run any `git` command except pushing. You need the smallest change that enforces this. What should you add?

A. `--excluded-tools='bash'`
B. A line in `AGENTS.md` saying "never run git push"
C. `--deny-tool='shell(git push)'`
D. `--allow-tool='shell(git:*)'` only, removing `--allow-all-tools`

Practice 6.3 MULTIPLE CHOICE

An auditor asks for every action that Copilot agents performed on behalf of the developer `mlopez` in the `contoso/claims-api` repository. You need a single enterprise audit log query. Which query should you use?

A. `actor:mlopez repo:claims-api`
B. `actor:Copilot user:mlopez repo:contoso/claims-api`
C. `user:Copilot actor:mlopez repo:contoso/claims-api`
D. `operation:create actor:mlopez repo:contoso/claims-api`

Practice 6.4 MULTIPLE CHOICE

Your enterprise streams its audit log to Azure Blob Storage for a five-year retention policy. During a storage migration the stream is paused for ten days and then resumed. You need to determine whether events were lost. What is the outcome?

A. No events are lost; the stream replays all ten days.
B. All events from the ten days are lost; the stream restarts from the current time.
C. The stream resumes from one week before the current time, so roughly the first three days of the pause are missing from the stream.
D. The stream cannot be resumed and must be recreated.

Practice 6.5 MULTIPLE CHOICE

Developers delegate infrastructure changes to Copilot cloud agent and then trigger the deployment workflow themselves. The `production` environment lists the platform team as required reviewers, and several developers are also members of that team. Compliance requires that the person who starts a production deployment cannot approve it. What should you configure?

A. Enable **Prevent self-reviews** on the `production` environment.
B. Add a wait timer of 60 minutes to the environment.
C. Add a `concurrency` group to the deployment job.
D. Disable **Require approval for workflow runs** in the Copilot cloud agent settings.

Practice 6.6 MULTIPLE CHOICE

A team configures content exclusion for `/secrets/**` at repository level and assumes this stops VS Code agent mode from reading those files during autonomous edits. You need to advise them. Which statement is correct?

A. The exclusion applies to agent mode after the window is reloaded.
B. The exclusion applies only if the paths are also added to a `.copilotignore` file.
C. The exclusion applies to agent mode only for Copilot Enterprise.
D. Content exclusion is not supported in agent mode, so they must keep those files out of the workspace or otherwise inaccessible.

Practice 6.7 MULTIPLE CHOICE

A product team complains that agent PRs which only change Markdown under `/docs` wait hours for review. The default branch ruleset requires one approval, code owner review and green status checks; a separate environment approval is required for every workflow run, including the docs link checker. You need to improve velocity without materially increasing risk. What should you do?

A. Remove the pull request rule from the default branch ruleset.
B. Remove the environment from the docs link-check job and rely on its required status check, keeping environment approval for production deployment jobs.
C. Disable Copilot's built-in code scanning to make the agent finish faster.
D. Add Copilot as a bypass actor on the default branch ruleset.

Practice 6.8 MULTIPLE CHOICE

A Teams channel conversation asks Copilot cloud agent to fix a bug, and Copilot opens the PR under its own app identity. The default branch ruleset requires one approving review and you have not changed any Copilot-specific ruleset settings. How many approvals does the PR need before it can merge?

A. Zero, because Copilot's commits are signed
B. One
C. Two
D. It can never merge because the agent cannot be a PR author

Practice 6.9 MULTIPLE CHOICE

An enterprise owner sets the enterprise default `GITHUB_TOKEN` permissions to the restricted setting. A developer asks which part of a Copilot cloud agent task this policy affects. Which answer is correct?

A. The token Copilot uses to push to its `copilot/` branch
B. The Agents secrets exposed to the session
C. The steps in the `copilot-setup-steps.yml` workflow
D. The agent firewall allowlist

Practice 6.10 MULTIPLE RESPONSE (CHOOSE TWO)

You are classifying actions a cloud agent may perform in a regulated payments repository. Which **two** actions should require explicit human authorisation before they take effect?

A. Adding unit tests for an existing validation function on the agent branch
B. Running a schema migration job that drops a column in the production database
C. Modifying `.github/workflows/release.yml` to add a new third-party action
D. Fixing linter warnings reported by a required status check

Answers and explanations

6.1: B. Hook timeouts are fail-open for every event, including `preToolUse` and admin policy hooks; the tool call proceeds through the normal permission flow with a warning. Fail-closed applies to crashes and non-zero exits, not timeouts (A). The cloud agent has no interactive user (C). A single hook timeout does not end the session (D). If the scanner must be authoritative, make it fast or perform the check in a way that cannot time out.

6.2: C. Deny rules always take precedence over allow rules, even with allow-all options, so `--deny-tool='shell(git push)'` blocks pushes while everything else stays pre-approved. Excluding `bash` (A) removes all shell use. Instructions (B) are guidance, not enforcement. Allowing only `shell(git:*)` (D) still permits `git push` and would leave other tools prompting in

an unattended run.

6.3: B. `actor:Copilot` selects agent-performed events, and for agent actions the `user` field holds the person the agent acted for. The `repo` qualifier must include the owner, so A is invalid; C swaps actor and user; D finds actions mlopez performed personally, not agent actions.

6.4: C. A paused stream keeps a seven-day buffer. After more than seven days it resumes from a point one week before the current time, so events older than that (about three days here) are not delivered. Only a pause of three weeks or more starts anew with no retained data (B).

6.5: A. Prevent self-reviews stops users who initiate a deployment from approving it even if they are required reviewers. A wait timer (B) delays but does not separate duties, concurrency (C) controls parallelism, and the workflow approval setting (D) concerns workflow runs on Copilot's pushes, and disabling it reduces control.

6.6: D. GitHub documents that content exclusion is not supported in the Edit and Agent modes of Copilot Chat in VS Code and other editors. Reloading (A) only refreshes settings on supported surfaces, there is no documented `.copilotignore` (B), and the plan (C) does not change agent-mode support. Real access controls are required.

6.7: B. The environment approval on a docs link checker adds delay without reducing risk; the required status check already enforces the outcome automatically. Keep human gates on the irreversible production step. Removing the PR rule (A) or adding a bypass (D) removes meaningful controls; disabling built-in scanning (C) trades security for speed.

6.8: C. With rulesets, "Require an additional approval for unattributed Copilot pull requests" is enabled by default (public preview), so a ruleset requiring one approval requires two when Copilot opens a PR under its own identity. Signing (A) is unrelated to approvals, and Copilot can author PRs (D).

6.9: C. GitHub's guardrails tutorial states that the enterprise `GITHUB_TOKEN` policy does not affect the token Copilot receives for its sessions, but the `GITHUB_TOKEN` is used in `copilot-setup-steps.yml`. Agents secrets (B) and the firewall (D) are configured separately.

6.10: B and C. Dropping a production column is irreversible and compliance relevant, so it belongs behind an environment with required reviewers. Changing a release workflow and introducing a third-party action changes a privileged execution path and the supply chain, so it should need code owner approval (and workflows on the agent's PR should wait for **Approve and run workflows**). Unit tests (A) and lint fixes (D) are low-risk, reversible and verified by checks and normal PR review.

Key takeaways

- Classify **actions**, not agents, along operational, security and compliance risk; irreversible actions, self-approval and changes to guardrails always need a human.
- Map each risk level to the lightest control that actually enforces the decision: automated checks for low risk, PR review for medium, CODEOWNERS and workflow approval for high, environment required reviewers for critical, hard blocks for prohibited.
- `preToolUse` is the only hook that can stop a tool before it runs. Any `deny` wins; `permissionDecisionReason` is required; command hooks fail closed on errors, every hook fails open on timeout; under the cloud agent `"ask"` means `"deny"`.
- In Copilot CLI, `--deny-tool` beats `--allow-tool` and even `--allow-all`; administrators can suppress allow-all flags with `permissions.disableBypassPermissionsMode`.
- Content exclusion is not an agent-mode or CLI boundary; the firewall covers only agent Bash processes; push protection stops secrets at push time.
- Least privilege: `permissions` at workflow level read-only, writes per job; Agents secrets only for what the agent needs; production credentials in protected environments; ephemeral runners.
- The cloud agent cannot approve or merge, pushes only to its own branch, and workflows on its pushes wait for **Approve and run workflows** by default.
- CODEOWNERS needs "Require review from Code Owners" to enforce; environments allow up to six required reviewers, one approval suffices, and "Prevent self-reviews" enforces separation of duties.
- Audit with `actor:Copilot`, `actor_is_agent`, `agent_session_id` and `user`; prefer specific `action:` events such as `artifact.destroy`.
- The audit log keeps 180 days (Git events seven days); stream to an external store for longer retention, and remember the seven-day pause buffer.

References

- Study guide for Exam GH-600: Developing in Agentic AI Systems
- Govern agents by risk (Microsoft Learn)
- Excessive Agency (Agents), Microsoft Learn
- Building guardrails for GitHub Copilot cloud agent
- Risks and mitigations for GitHub Copilot cloud agent
- About GitHub Copilot cloud agent
- GitHub Copilot hooks reference
- Using hooks with GitHub Copilot CLI
- GitHub Copilot CLI command reference
- Excluding content from GitHub Copilot
- Content exclusion for GitHub Copilot
- Customizing or disabling the firewall for GitHub Copilot
- Push protection
- Configuring settings for GitHub Copilot cloud agent
- Configure secrets and variables for Copilot cloud agent
- Configure the development environment (copilot-setup-steps.yml)
- Workflow syntax for GitHub Actions (permissions, environment)
- Use GITHUB_TOKEN for authentication in workflows
- Managing GitHub Actions settings for a repository
- Deployments and environments
- About code owners
- Available rules for rulesets
- REST API endpoints for rules
- Audit log events for agents
- Monitoring agentic activity in your enterprise
- Audit log for an enterprise

- Searching the audit log for your enterprise
- Audit log events for your enterprise
- Using the audit log API for your enterprise
- Streaming the audit log for your enterprise
- Secure AI-assisted development in VS Code
- Tool and command approvals in VS Code

CROSS-DOMAIN PRACTICE

Case study practice

A method for case studies and a full practice case: Fabrikam Payments

Case studies combine several domains in one scenario. Each item targets one requirement, and wrong options usually satisfy that requirement while breaking another one. Once you leave a case study section you cannot return.

C.1 A method for case studies



1. **Requirements first.** They contain the words that decide answers: "must not", "only", "without slowing", "retain", "approve".
2. **Classify each requirement by domain.** Tool scope (D2), state (D3), evaluation (D4), coordination (D5), guardrail (D6).
3. **Prefer platform controls over prompts.** When a requirement says an action *must never* happen, the answer is a control that enforces it (tool list, hook, ruleset, environment, permission), not an instruction asking the agent to behave.
4. **Check velocity.** An option that adds an approval to every low-risk action usually violates a "without slowing delivery" requirement.
5. **Review before leaving.** You cannot come back.

C.2 Case study: Fabrikam Payments

Overview

Fabrikam Payments runs 120 repositories in a GitHub Enterprise Cloud organisation. The platform team is rolling out GitHub Copilot cloud agent, custom agents, and Copilot CLI to 400 developers to accelerate a large framework upgrade and routine maintenance.

Current environment

- Builds depend on an internal package registry at `packages.fabrikam.internal`. Cloud agent sessions fail during dependency installation.
- The `payments-core` repository contains `.github/workflows/`, `infra/` (Terraform), and application code.
- Developers have added several community MCP servers to their VS Code configuration.
- A pilot agent "fixed" a failing build by editing a workflow to skip the test job.
- A large upgrade task timed out twice; the second session repeated steps the first had completed.
- Two agents upgraded different libraries in parallel and both edited `build/versions.props`.

Requirements

- (F1) A planning agent must analyse code and produce implementation plans, but must not edit files, run commands, or invoke other agents.
- (F2) Cloud agent sessions must be able to install packages from the internal registry.
- (F3) Developers may only use MCP servers approved by the security team.

- (F4) Changes to `.github/workflows/` and `infra/` must be approved by the platform team.
- (F5) A pull request may merge only when unit tests and CodeQL analysis pass.
- (F6) Production deployments must be approved by a release manager. Other environments must deploy without manual approval.
- (F7) Long-running upgrades must resume after interruption without repeating completed steps.
- (F8) Parallel agents must not produce conflicting changes to shared files.
- (F9) Records of agent activity must be retained for two years.

Requirement-to-control map

Requirement	Control	Why the tempting alternative fails	Domain
F1 read-only planner	Custom agent with `tools: ["read", "search"]` (plus any read-only MCP tools); no `edit`, `execute`, or `agent`.	An instruction saying "do not edit" is not enforcement.	D1, D2
F2 internal registry	Install dependencies in `copilot-setup-steps.yml` and allow the registry host through cloud agent's firewall configuration; supply credentials through the repository's Copilot environment secrets.	Disabling the firewall entirely widens exposure beyond the need.	D2
F3 approved MCP only	Organisation or enterprise MCP registry with a registry-only allowlist policy.	Asking developers to remove servers is not enforceable.	D2
F4 platform approval	CODEOWNERS for those paths plus a ruleset requiring code owner review.	A required review from "anyone" does not guarantee the platform team.	D6
F5 evidence to merge	Required status checks for the test job and CodeQL in a ruleset; agents prevented from weakening them (F4).	Agent self-reports in the PR description are not evidence.	D4
F6 gated production only	`production` environment with required reviewers; other environments have no reviewers.	Requiring approval for every deployment slows delivery without reducing risk.	D6
F7 resumable work	Progress file or PR checklist committed to the branch, updated after each step; resume prompt tells the agent to read it first.	Relying on chat history alone; sessions and context do not persist reliably.	D3
F8 no conflicts	One branch and PR per agent task, file ownership boundaries, a single owner for shared files, serialised changes to shared files.	Letting both PRs race and fixing conflicts manually at merge.	D5
F9 two-year records	Stream the enterprise audit log to long-term storage; keep session logs and artifacts with suitable retention.	Default audit log retention is shorter than two years.	D6

Practice items

Practice C.1 MULTIPLE CHOICE

You need to meet F1. Which `tools` value should the planning agent profile use?

A. `["*"]`
B. `["read", "search"]`
C. `["read", "search", "agent"]`
D. `["read", "edit"]`

Practice C.2 MULTIPLE RESPONSE

You need to meet F2. Which two actions should you perform? Each correct answer presents part of the solution.

A. Add the internal registry host to the cloud agent firewall allowlist
B. Disable the cloud agent firewall for the organisation
C. Restore dependencies in `copilot-setup-steps.yml` using credentials from the Copilot environment secrets
D. Commit the registry token to `.npmrc` in the repository
E. Increase the session timeout beyond 59 minutes

Practice C.3 MULTIPLE CHOICE

You need to meet F4 and prevent a repeat of the workflow edit that skipped tests. What should you configure?

A. A line in `copilot-instructions.md` asking agents not to edit workflows
B. CODEOWNERS entries assigning `.github/workflows/` and `infra/` to the platform team, and a ruleset that requires code owner review
C. Remove the `edit` tool from every agent
D. Disable Copilot cloud agent for `payments-core`

Practice C.4 YES/NO

For each statement about F6, select Yes if true, otherwise No.

A. A job that references the `production` environment waits for an approval from a required reviewer before it runs.
B. Adding required reviewers to the `staging` environment improves compliance with F6.
C. Environment protection rules apply even when the workflow was triggered by a pull request that an agent created.

Practice C.5 MULTIPLE CHOICE

You need to meet F7. What should the upgrade agent's instructions require?

A. Start every session from a fresh branch
B. Record each completed step and decision in a progress file committed to the working branch, and read that file before doing anything else when a session starts
C. Keep all progress in the chat conversation
D. Increase the model's context window

Practice C.6 MULTIPLE CHOICE

You need to meet F8 for future parallel upgrades. What should you do?

A. Let agents share one branch so they see each other's changes
B. Give each agent its own branch and pull request, assign shared files such as `build/versions.props` to a single owning task, and sequence changes to that file
C. Run agents one at a time across the whole organisation
D. Disable required checks while agents run

Practice C.7 MULTIPLE CHOICE

You need to meet F9. What should you implement?

A. Download session logs manually each month
B. Configure enterprise audit log streaming to a storage or SIEM destination with two-year retention
C. Increase Actions artifact retention to 90 days
D. Enable Copilot Memory

Answers and explanations

C.1: B. Read and search are enough to analyse code and produce a plan in the response. `*` grants everything, `agent` could delegate to an agent with forbidden tools, and `edit` violates F1.

C.2: A and C. Cloud agent's firewall limits outbound access, so the registry host must be allowed, and dependencies are best restored in setup steps with credentials from the Copilot environment. Disabling the firewall over-grants; committing tokens leaks secrets; the 59-minute session limit cannot be extended and is unrelated.

C.3: B. Code owners plus a ruleset that requires their review enforces platform approval. Instructions are advisory; removing `edit` everywhere stops all useful work; disabling the agent is disproportionate.

C.4: Yes, No, Yes. Required reviewers pause jobs that target the environment. Adding reviewers to staging adds friction without serving F6. Protection rules are evaluated for the job regardless of who opened the PR.

C.5: B. A durable, committed progress record lets a new session resume without repeating steps or diverging from earlier decisions.

C.6: B. Isolation by branch plus clear ownership and sequencing of shared files prevents overlapping changes while keeping parallelism for everything else.

C.7: B. Streaming exports audit events to a destination you control, where retention can exceed the platform default.

Key takeaways

- "Must never" requirements need enforcement: tool lists, hooks, rulesets, CODEOWNERS, environments, permissions.
- Gate only what is risky: production, workflows, infrastructure, credentials. Leave low-risk paths fast.
- State that must survive interruptions belongs in committed artifacts, not in chat history.
- Parallel agents need isolation (branch per task) and ownership of shared files.

EXAM DAY

Exam technique and signal words

Turning knowledge into points

GH-600 stems describe a risk or a goal and ask for the control or configuration that addresses it. These signal words map directly to answer families.

E.1 Signal words that map to answers

If the stem says...	Think first of...
Agent must plan but not change anything	Planner custom agent with read and search tools only; plan as Markdown output
Agree on the approach before a big change	Plan first (plan mode or planner agent), review the plan, then hand off to implementation
One-line issue leads to wrong scope	Well-scoped issue with context, acceptance criteria, and affected files
Agent must not run commands / must not delegate	Remove `execute` / remove `agent` from `tools`
Agent hidden from the picker but callable by other agents	`user-invocable: false`
Agent should only run when explicitly selected	`disable-model-invocation: true`
Use only specific tools from an MCP server	`server/tool` entries in `tools`, not `server/*`
Only approved MCP servers across the organisation	MCP registry plus allowlist policy
Pass a secret to an MCP server for cloud agent	Secret in the repository's Copilot environment, referenced from the MCP configuration
Cloud agent cannot reach an internal host	Firewall allowlist entry; dependencies in `copilot-setup-steps.yml`
Skip prompts for safe tools, keep prompts for risky ones	Allow-list specific tools; deny or require approval for destructive ones
Unattended run in a trusted sandbox or CI	Programmatic / autopilot run with explicit allow and deny rules and least-privilege token
Agent invoked from a workflow	Job-level `permissions` with only the scopes needed; workflow default read-only
Session approaching the context limit	Compact the conversation (keeps a summary); clear only when history is no longer needed
Resume without redoing work	Progress file or PR checklist committed to the branch; read it first
Share a plan between jobs	Job outputs for small values; artifacts for files
Stale or conflicting guidance	Single source of truth in instruction files; remove outdated instructions; validate memories against current code
Success = tests + security scan	Required status checks (unit tests, CodeQL) enforced by a ruleset
Agent edits workflows to pass checks	Protect workflow paths (CODEOWNERS + ruleset), treat as an evaluation-alignment failure
Session "succeeded" but PR is empty	Read the session log; classify as environment, tool, or reasoning failure

If the stem says...	Think first of...
Coordinator calls specialists	Orchestrator agent with subagents / `agent` tool and an explicit list of callable agents
Parallel agents, same files	Branch per task, file ownership, sequencing shared files, concurrency groups
Serialise per PR, parallel across PRs	Actions `concurrency` group keyed on the PR number
Replace or roll back an agent safely	Agent profiles versioned in git; change via PR; revert the commit
Block a dangerous command deterministically	Pre-tool hook that denies matching commands; CLI deny rules
Irreversible or compliance-sensitive change	Explicit human authorisation: environment reviewers, code owner review, approval step
Keep Copilot away from secrets or templates	Content exclusion
Who did it, and was it an agent?	Audit log search by actor and action; agent session records; commit attribution
Keep records beyond default retention	Audit log streaming to your own storage

E.2 Distractor patterns

Pattern	Example
Advice instead of enforcement	"Add a line to the instructions telling the agent not to..." when the requirement says "must never".
Over-broad grant	`tools: ["*"]`, `permissions: write-all`, disabling the firewall, Owner role.
Approval everywhere	Required reviewers on every environment or every tool call when only one path is risky.
Chat history as state	Relying on the conversation to remember progress across sessions or jobs.
Treating the symptom	Retrying a timed-out session with the same scope; deleting logs; disabling a failing check.
Wrong layer	Using Copilot Memory to share state between Actions jobs; using artifacts to restrict tools.

E.3 Time management

- About two minutes per item; keep 15 minutes for case studies and review.
- For configuration items, locate the blank and the one requirement it serves before reading every line.
- Mark anything below 70% confidence and move on; never leave an item blank.
- When two options both work, pick the one that is enforceable and least privileged and that adds the fewest approvals.

APPENDIX A

One-page cheat sheet

Decisions you should make in under ten seconds

Decision	Rule
Where should the agent run?	**Cloud agent**: delegated, async, PR output, one repo, one branch, 59-minute session cap. **VS Code agent mode**: interactive local work. **Copilot CLI**: terminal or CI, scriptable, allow/deny rules. **Code review**: PR feedback.
Planning vs action	Separate them: planner agent (read/search) produces a structured plan; a human or validation step approves; implementer agent (edit/execute) acts; handoff between them.
Tools	Grant the minimum aliases (`read` , `search` , `edit` , `execute` , `agent` , `web`); MCP tools as `server/tool` . `[]` = none, omitted or `*` = all.
Agent visibility	`user-invocable: false` hides from picker; `disable-model-invocation: true` prevents automatic selection. `infer` is retired.
MCP	VS Code: `.vscode/mcp.json` . Cloud agent: repository MCP settings, secrets in the Copilot environment. Agent profile: `mcp-servers` . Govern with a registry and allowlist policy. GitHub remote MCP server: choose toolsets, prefer read-only where possible.
Environment	`copilot-setup-steps.yml` for dependencies, runner, and `timeout-minutes` ; firewall allowlist for outbound hosts; secrets and variables in the Copilot environment.
CI invocation	Workflow default `permissions` read-only; grant `contents` / `pull-requests` write only on the job that needs it; explicit tool allow/deny for the CLI.
Errors	Retry transient failures with limits; roll back with `git revert` ; escalate to a human with context; record everything in logs and the PR.
Memory	**Short-term**: session context (compact when near limit). **Long-term**: Copilot Memory (repository-scoped, validated against current code), instruction files. **External**: committed files, issues, PRs, artifacts.
State between jobs	Job outputs for small strings; artifacts for files; repository files or PRs for durable, reviewable state.
Evaluation signals	Quantitative: tests, check status, CodeQL alerts, merge rate, iterations, time to merge. Qualitative: review comments, plan quality, adherence to instructions.
Root-cause classes	**Reasoning**: wrong plan or misread task. **Tool misuse**: wrong tool, bad parameters, missing permission. **Context/environment**: missing dependency, firewall block, stale or conflicting instructions, timeout.
Multi-agent patterns	Sequential handoff; orchestrator with subagents; fan-out/fan-in; per-item parallel with concurrency groups.
Conflicts	Overlapping edits: branch isolation and file ownership. Duplicated effort: single task registry (issues) and assignment. Contradictory outputs: designated reviewer or arbiter agent plus human decision.
Lifecycle	Agent profiles in git; changes by PR; roll back by revert; retire by removing from orchestrator lists while keeping history and logs.
Autonomy	Classify actions by operational, security, and compliance risk. Low risk: autonomous. Medium: autonomous with checks and review. High or irreversible: explicit human authorisation.
Guardrails	Hooks (deterministic block/log), CLI deny rules, content exclusion, CODEOWNERS, rulesets, required checks, environments with required reviewers, least-privilege tokens.

Decision	Rule
Accountability	Session logs, commit and PR attribution, Actions logs and artifacts, enterprise audit log (search and streaming for long retention).

APPENDIX B

Glossary

Terms used across the domains

Agent profile	A `.agent.md` file defining a custom agent: YAML front matter (description, tools, model, MCP servers, invocation settings) and a Markdown prompt.
Agent skill	Reusable, task-specific instructions and resources that agents load when relevant.
Audit log	Organisation or enterprise record of events, including agent-related activity; searchable and streamable.
Autonomy level	How much an agent may do without human confirmation, set per action class according to risk.
Autopilot	Copilot CLI mode for unattended runs within configured permissions.
CODEOWNERS	File assigning owners to paths; with rulesets or branch protection, requires their review.
Compaction	Summarising conversation history to free context while keeping key facts.
Content exclusion	Setting that prevents Copilot from using specified files or paths.
Context drift	Gradual divergence of an agent from the task, prior decisions, or current code during long execution.
Copilot cloud agent	GitHub's autonomous agent that works in an ephemeral cloud environment and delivers changes as pull requests (formerly Copilot coding agent).
Copilot Memory	Persistent, repository-scoped knowledge Copilot builds across sessions, with validation against current code.
`copilot-setup-steps.yml`	Workflow file that prepares cloud agent's environment before it starts work.
Environment (Actions)	Deployment target with protection rules such as required reviewers and wait timers.
Fan-out / fan-in	Running independent subtasks in parallel and merging their results in a later step.
Handoff	A transition from one custom agent to another (for example planner to implementer), usually with context and a user confirmation.
Hook	A command run at an agent lifecycle event (for example before a tool executes) to validate, block, or log.
MCP	Model Context Protocol: standard for connecting agents to external tools and data servers.
MCP registry / allowlist	Curated list of approved MCP servers and the policy that restricts usage to it.
Orchestrator	An agent that plans work and delegates subtasks to subagents.
Ruleset	Repository or organisation rules governing branches and tags: required reviews, status checks, restrictions.
Session log	Record of an agent session's reasoning steps, tool calls, and outputs.
Subagent	An agent invoked by another agent with its own context to perform a subtask.
Toolset	A named group of tools, for example GitHub MCP server toolsets.

APPENDIX C

Hands-on lab plan

Ten labs that cover every domain

#	Lab	Steps	Domain
1	Scoped delegation	Write an issue with context and acceptance criteria; assign it to Copilot cloud agent; iterate via PR comments; inspect the session log.	D1
2	Plan then act	Create planner and implementer custom agents; plan a change, review the plan, hand off, and compare the PR to the plan.	D1
3	Tools and MCP	Restrict tools in a profile; add the GitHub remote MCP server with selected toolsets in VS Code; add a repository MCP server with a secret for cloud agent.	D2
4	Environment	Write `copilot-setup-steps.yml`; trigger a firewall block and fix it with an allowlist entry.	D2
5	CLI in CI	Run Copilot CLI from a workflow with read-only defaults, job-level write permission, and explicit allow/deny rules.	D2, D6
6	State and memory	Enable Copilot Memory; maintain a progress file; pass a plan between jobs with outputs and artifacts; compact a long CLI session.	D3
7	Evaluation	Require tests and CodeQL; add code review instructions; break a task deliberately and classify the root cause from logs.	D4
8	Multi-agent	Orchestrator with two subagents; run two tasks in parallel on separate branches; add a concurrency group keyed on PR.	D5
9	Guardrails	Add a pre-tool hook that blocks destructive commands; content exclusion for secrets templates; CODEOWNERS and a ruleset for workflows.	D6
10	Accountability	Protect production with an environment reviewer; query the audit log for agent activity; configure streaming if available.	D6

APPENDIX D

Final readiness checklist

Tick every line before booking

☐	I can...
☐	Choose the right surface (cloud agent, VS Code, CLI, code review) and state its limits.
☐	Write a well-scoped issue with inputs, outputs, and success criteria.
☐	Separate planning from execution and require plan approval before action.
☐	Write an agent profile with least-privilege tools, invocation settings, and MCP tools.
☐	Configure MCP servers in VS Code, for cloud agent, and through registry and allowlist policies.
☐	Prepare the cloud agent environment, firewall, secrets, and timeouts.
☐	Invoke an agent from a workflow with least-privilege permissions.
☐	Implement retries, rollbacks, and escalation paths with traceability.
☐	Choose short-term, long-term, or external memory and manage expiry, pruning, and drift.
☐	Define evaluation signals and enforce them with required checks and scanning.
☐	Classify root causes from logs and tune instructions, memory, and tools.
☐	Apply an orchestration pattern, isolate parallel agents, and resolve conflicts.
☐	Add, update, roll back, and retire agents without breaking workflows or audit trails.
☐	Assign autonomy levels by risk and implement hooks, content exclusion, CODEOWNERS, rulesets, and environments.
☐	Find and retain records of agent activity.
☐	Score 85% or better on every chapter's practice questions.

APPENDIX E

Official resources

Where to go next

- GH-600 study guide (skills measured)
- Foundations of Agentic AI in GitHub
- Designing Agent Architecture and SDLC Integration
- Tooling, MCP, and Agent Execution Environments
- GitHub Copilot documentation
- Custom agents configuration
- About GitHub Copilot Memory
- Risks and mitigations for Copilot cloud agent
- VS Code custom agents
- Exam sandbox

Domain chapters list the specific documentation used for each topic.