MockCert

Study guide | GitHub | GH-600

GitHub Certified: Agentic AI Developer

Notes for every domain of the GitHub Certified: Agentic AI Developer exam, organized by the official exam guide's domains and weights.

By MockCert about 89 min read Updated November 23, 2026

CONTENTS

DOMAIN 1 · 15–20%
Prepare agent architecture and SDLC processes 2

DOMAIN 2 · 20–25%
Implement tool use and environment interaction 12

DOMAIN 3 · 10–15%
Manage memory, state, and execution 24

DOMAIN 4 · 15–20%
Perform evaluation, error analysis, and tuning 32

DOMAIN 5 · 15–20%
Orchestrate multi-agent coordination 42

DOMAIN 6 · 10–15%
Implement guardrails and accountability 53

MockCert is an independent certification preparation platform. It is not affiliated with, endorsed by or operated by GitHub. Exam domains and weights follow the official exam guide. These notes were last updated on November 23, 2026; the notes at mockcert.com are kept up to date as the exam changes.

Domain 1 | 15–20% of the exam | 16 min read

Prepare agent architecture and SDLC processes

Overview

Every point below comes from GitHub's official documentation; each section lists its sources.

Integrate agents into the software development lifecycle (SDLC)

Start here

THINK FIRST

How do you hand a task to an agent so its result is safe to merge and actually solves the problem?

Why it matters: Questions describe an agent task and ask which steps, inputs, success criteria or mitigations fit it.

An agent joins your workflow as a participant; it does not replace it. You define each task by its inputs, outputs and success criteria. The agent works on a branch and opens a pull request, which waits for review and validation before it merges.

Key terms

Term	Meaning
Agent	A system that uses tools to produce branches, commits and pull requests, and iterates on checks, reviews and scans.
Task contract	The definition of an agent task: what it needs (inputs), what it produces (outputs) and how results are judged.
Required check	A CI workflow set through rulesets or branch protection that must pass before a pull request can merge.
Plan agent	A mode that researches with read-only tools and makes no code changes until you approve its plan.
Overreliance	Accepting incorrect or incomplete AI output, often because its mistakes are hard to spot.

Agents join the workflow

An agent is a participant in the workflow, not a replacement for it. An assistant is reactive: it waits for the user to choose the next step. An agent uses tools such as the GitHub API, CI workflows or repository writes. It produces lasting results like branches, commits and pull requests, then iterates on feedback.

The pull request flow

Agent work moves through a branch and a pull request. The agent commits its proposed changes to a branch and opens a pull request. It then waits for review and validation before any merge. Assigning an issue to Copilot always creates a pull request. To ask for a fix, mention @copilot in a comment, and it pushes new commits to that same branch.

EXAM TRAP

Wrong: the agent merges its own work once it is done. Copilot has a default guardrail that stops it from merging pull requests, and rulesets can add more protections for target branches.

Steps agents can perform

Hand agents maintenance, feature and background work. Maintenance includes security fixes, dependency upgrades and targeted refactoring. Feature work includes incremental feature requests, extra test suites and new or updated docs. Agents can also run tests or fix backlog issues asynchronously, with less human help, and support ideation or post-release optimization.

Plan before coding

The plan agent researches first and changes no code until you approve. It uses read-only tools to find requirements and constraints. It splits the task into actionable steps and lists open questions about unclear requirements. Once the plan is ready, Start Implementation switches Copilot Chat to agent mode to build it.

Inputs: a well-scoped issue

Clear, well-scoped inputs give the best agent results. An ideal issue has a clear problem statement, complete acceptance criteria and hints about which files to change. The Optional prompt field can add coding patterns, files to modify or testing requirements. Copilot reads the issue title, description and existing comments when you assign it.

EXAM TRAP

Wrong: add new details as a fresh comment on the issue. Copilot does not see issue comments posted after assignment, so put follow-up details on the pull request.

The task contract

Define every agent task by inputs, outputs and success criteria. In the example contract, the outputs are a structured plan in the pull request description and a bounded changeset on an agent branch. Evidence links point to the workflow run.

The three parts of an agent task

Feature	Inputs	Outputs	Success criteria
Answers	What the agent needs	What the agent produces	How results are evaluated
Examples	Issue context, constraints, boundaries	Plan, pull request, evidence	Checks, scans, review outcomes

Success criteria that match intent

A required check lets the system enforce success. With the CI workflow set as a required check, the pull request cannot merge until it passes. Criteria should also confirm the real goal, such as "vulnerability resolved". Check that scope matches intent, with no unexpected files changed. Record a rollback or escalation path for higher-risk changes.

EXAM TRAP

Wrong: "tests passed" proves the task is done. Passing CI is necessary but not always enough; the criteria must reflect what the task was really meant to achieve.

Anti-patterns and fixes

Watch for vague tasks, overreliance, broad access and needless computer use. The table pairs each one with a mitigation. Match the mitigation to the risk that a question describes.

Common agent anti-patterns and how to mitigate them

Anti-pattern	What goes wrong	Mitigation
Under-specified task	Plausible changes that miss the real problem, like a vulnerable version still reachable transitively	A well-scoped issue with complete acceptance criteria
Overreliance on AI	Incorrect or incomplete output is accepted because errors are hard to detect	Use Copilot code review to supplement human review
Broad access	A session gets more tools and permissions than the task needs	Scope each SDK session to the tools, agents and permissions required
Computer use when a direct tool exists	Less structured information and less predictable results	Use an API, MCP server, terminal command or filesystem tool

EXAM TRAP

Wrong: AI code review can replace human reviewers. Copilot code review should supplement human review, since mistakes in AI output can be hard to detect.

Key takeaways

- An agent participates through branches and pull requests, and Copilot cannot merge its own pull requests.

- Define each task by inputs, outputs and success criteria, and make the criteria reflect the real intent.
- Make CI a required check so the system, not the agent, enforces success.
- Fix vague tasks with a clear problem, complete acceptance criteria and hints about which files to change.
- Give each session only the tools and permissions its task needs.

Sources: Define agentic AI in the SDLC - Training | Microsoft Learn (opens in a new tab) · Define inputs, outputs, and success criteria - Training | Microsoft Learn (opens in a new tab) · Using plan mode in your IDE - GitHub Docs (opens in a new tab) · Application card: GitHub Copilot Agents - GitHub Docs (opens in a new tab) · Kick off a task with Copilot agents on GitHub - GitHub Docs (opens in a new tab) · Get started with Copilot agents on GitHub - GitHub Docs (opens in a new tab) · Understanding how AI agents fit into your enterprise's workflows - GitHub Docs (opens in a new tab)

Define boundaries between planning, reasoning, and action

Start here

THINK FIRST

How do you make sure an agent's plan is checked and approved before the agent is allowed to change anything?

Why it matters: Questions describe an agent task and ask how to separate planning from action, where the plan appears, or which gate stops execution.

Planning and action are separate stages with a deliberate handoff between them. The plan is an artifact a human can review. Tool limits and approval gates enforce the pause; written instructions only guide it.

Key terms

Term	Meaning
Plan-first	The agent opens a pull request holding only its plan, so a human validates it before code exists.
Tool gating	An orchestrator runs planning with no tool execution and enables tools only once the plan is accepted.
Safe outputs	The agent proposes actions, and a separate gated step decides which are allowed.
Protection rule	An environment check, such as required reviewers or a wait timer, that must pass before a job starts.
Actor rules	Rules that separate who can write code from who can trigger workflows.

Keep planning separate from action

A planning agent should get read-only tools only. That way it cannot modify files while it plans. Execution should start only after the plan is approved, through an explicit handoff to an implementation agent. Plan mode in some interfaces produces a plan artifact and pauses before applying changes. In orchestrations,

tool gating gives the same pause.

> **EXAM TRAP**
> Telling an agent "do not edit files" is not a control. Instructions are guidance; tool allowlists and gates are what enforce the boundary.

Where a plan appears

Plans are written down; execution shows up as commits. A structured plan can sit in a pull request description, an issue comment or a `pull_request_template.md` artifact. Code changes appear as commits on a branch. Two pull request patterns follow from this. Use plan-first for high-risk work such as workflows, infra, auth or production.

Plan-first and plan-plus-execution pull requests compared

Feature	Option A: Plan-first	Option B: Plan + execution
What the pull request holds	Only the plan, no code changes yet	A structured plan in the description plus initial commits
When a human validates	Before any code is written	After code is generated, before merge

Validating the plan and its result

Review rests on evidence: checks run and human approvals recorded. Workflows and checks show what evidence exists. Review comments and approvals show what people accepted or rejected. A plan can also set scope, such as changes allowed under `src/` but not `infra/` unless requested. Copilot in the cloud prepares changes for review but does not automatically create or merge the pull request.

> **EXAM TRAP**
> If approvals are required, your own approval of a Copilot pull request does not count. Another reviewer must approve before it can merge.

Escalating to a human

Escalation lets a person step in when the agent is stuck or unsure. You can build it by requiring pull request review, assigning reviewers automatically or notifying maintainers from a workflow step. Note that by default, Actions workflows do not run on their own when Copilot pushes to a pull request.

Approval gates on environments

A job that uses an environment waits until every protection rule passes. Until then it cannot read the environment's secrets either. A waiting job shows the status "Waiting" and fails automatically after **30** days without approval. Once approved, and with other rules passed, it proceeds.

Settings that tighten an environment approval gate

Setting	What it does	Detail
Required reviewers	A reviewer must approve the job	Up to **6** people or teams; one approval is enough
Prevent self-review	Blocks approving a run you triggered	Optional
Allow administrators to bypass	Lets admins skip the rules	Deselect it to disallow bypassing

EXAM TRAP

On GitHub Free, Pro or Team plans, required reviewers and wait timers are available only for public repositories.

Environment behavior to watch

Some environment setups quietly remove or break the gate. Custom protection rules are GitHub Apps that approve or reject jobs from the `deployment_protection_rule` webhook. They need a deployment object.

What happens to the gate in each situation

Situation	Effect on the job	Gate still applies?
`deployment: false`	Full access to secrets; no deployment object or history	Yes: wait timers and required reviewers
`deployment: false` with custom rules	Fails immediately with an error	Not applicable
Workflow names an environment that does not exist	A new environment is created	No rules or secrets, unless made by a page build
Environment is deleted	Its secrets and rules are deleted	Waiting jobs fail automatically

Who may run automation

Limit who and what can execute, not only what the agent plans. By default, anyone with write access can trigger workflows; actor rules separate contributing code from running CI. Follow least privilege for workflow permissions. Prefer GitHub Agentic Workflows over calling Copilot CLI in run steps, since they carry guardrails for automation. Require lockfile protection blocks targeted workflows not onboarded to the lockfile.

Key takeaways

- Give the planning agent read-only tools and hand off to an implementation agent only after approval.
- Plan-first puts human validation before any code; plan plus execution validates before merge.
- Instructions are guidance; allowlists, tool gating and approval gates are enforcement.
- An environment job waits for its protection rules and fails after **30** days without approval.

Sources: Separate planning, reasoning, and execution - Training | Microsoft Learn (opens in a new tab) · Execution context and boundaries - Training | Microsoft Learn (opens in a new tab) · Deploying with GitHub Actions - GitHub Docs (opens in a new tab) · Managing environments for deployment - GitHub Docs (opens in a new tab) · Governing execution of GitHub Actions workflows - GitHub Docs (opens in a new tab) · Define agentic AI in the SDLC - Training | Microsoft Learn (opens in a new tab) · Define inputs, outputs, and success criteria - Training | Microsoft Learn (opens in a new tab) · Agent execution limits and protections - Training | Microsoft Learn (opens in a new tab)

Configure observability and control for autonomous agents

Start here

THINK FIRST

How can an autonomous coding agent work freely while people can still see, steer, and stop what it does?

Why it matters: Questions describe an agent setup and ask which guardrail, log, hook, or human approval point controls it.

Let the agent work on its own inside fixed guardrails, make every action visible, and keep people at the merge and deploy gates. Copilot cloud agent pushes to its own branch, logs its reasoning, and needs a human to merge.

Key terms

Term	Meaning
Hook	Code that runs at set points while the agent works, enforcing a rule instead of asking the model to follow it.
Session log	What Copilot reasoned and which tools it ran to read, change, and validate code.
Steering	Sending a prompt while Copilot is working so it is folded into the current task.
copilot environment	The environment whose secrets and variables are the only ones passed to Copilot cloud agent.
Required reviewers	An environment setting that makes a workflow job wait until a person approves it.

Default guardrails

Copilot cloud agent can push to only one branch. On an existing pull request it writes to that pull request's branch; otherwise it gets a new copilot/ branch. It cannot push to the default branch or merge, and rulesets apply to it as to people. Only users with write access can start it.

EXAM TRAP

Comments from users without write access do not influence the agent. They are never shown to it at all.

Secrets, runners, and built-in checks

The agent sees only secrets placed in the copilot environment. Organization and repository Actions secrets stay hidden from it. GitHub-hosted runners give each session a fresh virtual machine; self-hosted setups should use ephemeral runners. A firewall blocks data exfiltration, and by default the agent scans its code for security issues and asks Copilot code review for a second opinion.

EXAM TRAP

The agent's security validation does not need a GitHub Advanced Security, Code Security, or Secret Protection license.

Hooks enforce rules with code

Hooks run code at key points, so rules are enforced, not requested. Use them for linting, formatting, or security scans. Repository hooks live in `.github/hooks/*.json`; Copilot CLI also reads personal hooks from `~/.copilot/hooks/*.json`. Hooks block the agent while they run, so aim for under **5 seconds**. The cloud agent supports only some hook events.

The two main tool hooks compared

Feature	preToolUse	postToolUse
When it runs	Before the agent uses any tool, such as bash or edit	After a tool finishes, whether it worked or failed
Main use	Approve or deny the tool call	Log results and build audit trails

Permissions in Copilot CLI

Copilot CLI keeps file access inside the session's allowed paths. Going outside those paths needs extra permission. With assisted permissions, a request that still needs approval prompts you interactively but is denied in autopilot and other unattended runs. The computer-use plugin is off until a user enables it, explicit deny rules win, and unhandled requests fail closed.

Inspectable artifacts

Each agent commit links back to the session that made it. Reviewers can trace any change back to the session that made it. Administrators also get session logs and audit log events.

Where to inspect what the agent did

Artifact	What it shows	Why it helps
Commit	Signed and Verified; authored by Copilot with the requesting developer as co-author	Shows who asked for the change
Commit message	A link to the agent session logs	Supports code review and auditing
Session log	Copilot's reasoning and the tools it used	Explains how the change was made and checked
Agents tab	All sessions, shared by default	Anyone with repository access can view them
Diff and copilot/ branch	The changes, alone or in context	Review before a pull request exists
Workflow run graph	Live progress of the run	Monitor and debug deployments

Human review gates

A human must review and merge the agent's draft pull requests. Copilot cannot mark them ready for review, approve, or merge them. Workflows on its code wait until a write-access user clicks Approve and run workflows, unless you allow automatic runs. Protect Copilot and MCP configuration files with CODEOWNERS and the Require review from Code Owners rule.

EXAM TRAP

The person who asked the agent for a pull request cannot approve it. When approvals are already required, the agent's pull request also needs one more approval.

Steer without stopping

Steering redirects a session without halting it. Copilot applies your input once its current tool call ends. Each steering message uses AI credits, and third-party coding agents cannot be steered. Stopping a session ends its Actions run but keeps commits already pushed. When you ask the cloud agent for a plan, review it and iterate until it matches your intent.

Steering and queuing a prompt in Copilot CLI

Feature	Steer	Queue
How to send	Enter; a plain prompt sent while Copilot thinks	Ctrl+Enter or Ctrl+Q
When it applies	Within the task in progress	After the current task finishes
Effect	Joins the task in progress	Runs as the next turn

Gate deployments, keep speed

Environments with required reviewers pause risky jobs. A job aimed at such an environment waits for approval, so agent-triggered workflows cannot reach protected secrets or deploy alone. To avoid delays, let services like Datadog or ServiceNow approve automatically, and get app notifications when a deployment

awaits approval. Concurrency can allow just **1** deployment at a time.

Key takeaways

- Copilot cloud agent pushes to a single branch and cannot approve or merge its own pull requests.
- A preToolUse hook can approve or deny tool use; a postToolUse hook can build audit trails.
- Each agent commit links to its session logs, which show Copilot's reasoning and tools.
- Steering applies after the current tool call; a queued prompt waits until the task finishes.
- By default, workflows on agent code wait until a write-access user approves and runs them.

Sources: Risks and mitigations for GitHub Copilot cloud agent - GitHub Docs (opens in a new tab) · Managing agent sessions - GitHub Docs (opens in a new tab) · Describe GitHub as the system of record and control plane - Training | Microsoft Learn (opens in a new tab) · Hooks for GitHub Copilot - GitHub Docs (opens in a new tab) · Customize agent workflows with hooks - GitHub Docs (opens in a new tab) · Steering agents in GitHub Copilot CLI - GitHub Docs (opens in a new tab) · Application card: GitHub Copilot Agents - GitHub Docs (opens in a new tab) · Building a secure cloud environment for agents in your enterprise - GitHub Docs (opens in a new tab)

Domain 2 | 20–25% of the exam | 18 min read

Implement tool use and environment interaction

Overview

Every point below comes from GitHub's official documentation; each section lists its sources.

Select and configure agent tools

Start here

THINK FIRST

How do you give an agent the tools a task needs while keeping risky actions under your control?

Why it matters: Questions describe a task or a tool setup and ask which tools, configuration, or permission option fits.

Choosing tools means giving an agent only what the task needs. MCP servers and their toolsets add abilities, and a repository can share one MCP configuration. Permission settings then decide which tools run freely, which ask first, and which are blocked.

Key terms

Term	Meaning
MCP server	A server that supplies an agent with tools, like the GitHub MCP server.
Toolset	A named group of GitHub MCP server tools, such as repos, issues, or pull_requests.
Repository MCP configuration	Repository settings that list MCP servers and their tools, shared by cloud agent and code review.
Available set	The tools the model can use at all in Copilot CLI, chosen with `--available-tools`.
Deny rule	A rule that blocks a tool; it beats any allow rule or saved approval.

Matching the mode to the task

Use edit mode for tight control and agent mode for automation. In edit mode you pick the files Copilot may change and accept or reject each edit. Agent mode handles repetitive workflow work, like opening pull requests or running tests and linters in the background.

Copilot Chat edit mode and agent mode

Feature	Edit mode	Agent mode
Best for	Granular control over changes	Automating repetitive workflow tasks
What you decide	Which files change and whether to accept each edit	Which tasks to hand off while you keep working
Typical work	Iterating on chosen files with added context	Creating pull requests, running tests and linters

GitHub MCP server toolsets

The repos, issues, and pull_requests toolsets are on by default. Some toolsets, such as copilot and github_support_docs_search, exist only on the remote server. Removing `/readonly` from the remote server URL widens access to all tools.

Remote and local GitHub MCP server setup

Feature	Remote server	Local server
Selecting toolsets	List them in the `X-MCP-Toolsets` header	Use command-line flags or environment variables; variables win
Remote-only toolsets	copilot and github_support_docs_search available	Not available

EXAM TRAP

On a local server, a command-line flag does not win over an environment variable. Environment variables take precedence over flags.

Repository MCP configuration

One repository MCP setup serves cloud agent and code review. Cloud agent, including its automations, shares it with Copilot code review. A local server needs a `command` that starts it and the `args` passed to that command. String fields other than `tools` and `type` can pull in a Copilot environment variable or secret.

Porting and verifying servers

Reusing a VS Code setup needs a `tools` key for each server. Swap any `inputs` or `envFile` for `env`. Give GitHub App values as Agents secrets prefixed `COPILOT_MCP_`. Install extra dependencies, such as uv or pipx, in a `copilot-setup-steps.yml` workflow. Then expand the Start MCP Servers log step to see each server's tools.

EXAM TRAP

Storing those values as variables is the wrong choice. Only secrets stay available when Copilot runs in a cloud sandbox.

Scoping tools for reviews and automations

Code review may call MCP tools by default; you can turn that off. Disabling the setting stops code review from calling any MCP tool, even the default GitHub and Playwright servers. An automation can only use the MCP tools picked when it was created or edited. Organizations and enterprises can enforce MCP allowlists tied to each user's Copilot seat.

EXAM TRAP

Copilot connectors skip repository MCP setup in cloud sandbox sessions. They do not yet work in sessions on GitHub Actions, such as one started by assigning an issue.

Limiting the available tools

`--available-tools` **keeps only the tools you list.** By contrast, `--excluded-tools` turns off just the tools you name. When you pass both, Copilot CLI uses the allowlist and ignores the denylist.

Two ways to narrow Copilot CLI tools

Feature	--available-tools	--excluded-tools
What it turns off	Every tool you did not list	Only the tools you list
When both are passed	Applied as the allowlist	Ignored

EXAM TRAP

Adding `--allow-tool` cannot rescue a tool outside the available set. The model still cannot use it at all.

Allow and deny rules

Deny rules always beat allow rules. Read-only actions like searching and reading files are generally allowed automatically; edits, destructive commands, and URL access need permission. `--allow-tool` lets a tool run without a prompt, and `--deny-tool` blocks it. Pairing `--allow-tool='shell(git:*)'` with `--deny-tool='shell(git push)'` permits every git command except git push. `--allow-all`, or `--yolo`, equals `--allow-all-urls`, `--allow-all-tools`, and `--allow-all-paths` combined.

EXAM TRAP

Using `--allow-all` does not unlock a denied tool. Deny rules still win, even over a saved approval.

Saved approvals and permission modes

Approvals for a location persist; command-line options do not. Approving a tool for the current location saves it to `~/.copilot/permissions-config.json` by default. `--allow-tool` and `--deny-tool` last only for the session. `/reset-allowed-tools` revokes grants from the session. With `/permissions assisted`, a language model approves eligible requests it judges safe, but enterprise rules still apply.

Key takeaways

- Deny rules win over allow rules, even with `--allow-all` or a saved approval.
- With both `--available-tools` and `--excluded-tools`, only the `--available-tools` allowlist applies.
- The GitHub MCP server turns on the repos, issues, and pull_requests toolsets by default.
- Each server in a repository MCP configuration needs a `tools` key naming the tools Copilot may use.
- An automation can call only the MCP tools picked when it was created or edited.

Sources: Configuring toolsets for the GitHub MCP Server - GitHub Docs (opens in a new tab) · Configure MCP servers for your repository - GitHub Docs (opens in a new tab) · Allowing and denying tool use - GitHub Docs (opens in a new tab) · Choosing the right AI tool for your task - GitHub Docs (opens in a new tab) · Model Context Protocol (MCP) servers, registries, and allow lists - Training | Microsoft Learn (opens in a new tab)

Configure MCP servers

Start here

THINK FIRST

How do you give an agent tools from an MCP server while still controlling which servers your developers are allowed to run?

Why it matters: Questions describe an MCP setup and ask which setting, transport, registry policy or allowlist makes a server work or blocks it.

An MCP server hands an agent tools, and the agent then uses them without asking. You add servers per client or agent. The enterprise decides which servers may run, through a registry policy or, more securely, an allowlist in managed settings.

Key terms

Term	Meaning
MCP registry	A server list that follows the MCP registry v0.1 specification and exposes `GET /v0.1/servers`.
Registry only	A policy that lets only servers from the configured registry run, local ones included.
MCP allowlist	Allowed and denied servers set in `managed-settings.json`, matched by name, server URL or command.
Streamable HTTP	The transport behind the `http` server type; `sse` is the older, deprecated one.

Tools run without approval

Once a server is configured, Copilot uses its tools autonomously. No approval prompt appears before a tool call. So allowlist specific read-only tools; `*` in the tools array enables every tool. Copilot cloud agent

and Copilot code review use only MCP tools, not resources or prompts. Code review also skips any tool whose `annotations.readOnlyHint` is missing or `false`.

EXAM TRAP

Expecting Copilot to ask before each MCP tool call is wrong. It acts on its own, which is why you list read-only tools instead of `*`.

Adding a server in Copilot CLI

`copilot mcp add` **adds a server without interactive mode.** The server lands in the user configuration, `~/.copilot/mcp-config.json`. The `--transport` option takes `stdio`, `http` or `sse`, and `stdio` is the default. Project-level servers load only after you confirm folder trust. When `.mcp.json` and `.github/mcp.json` share a directory, `.mcp.json` wins.

The three transports for a new MCP server in Copilot CLI

Transport	Detail	Status
`stdio`	Also accepted by Copilot cloud agent	Default for `copilot mcp add`
`http`	Streamable HTTP	Current HTTP transport
`sse`	HTTP with Server-Sent Events	Deprecated, still supported for backwards compatibility

Built-in servers and secrets

The GitHub and Playwright MCP servers are on by default in cloud agent. You add your own servers next to them. The default GitHub server uses a token with read-only access to the current repository. Only Agents secrets and variables named with the `COPILOT_MCP_` prefix reach your MCP configuration. In Copilot CLI, the GitHub MCP server is built in and needs no setup.

EXAM TRAP

A secret named `API_KEY` is not visible to the MCP configuration. Rename it with the `COPILOT_MCP_` prefix.

The remote GitHub MCP server

In supported IDEs, remote setup is the recommended choice for most users. To add it, pick HTTP as the server type and enter the GitHub MCP server URL. GitHub Enterprise Server supports local configuration. With data residency, GitHub Enterprise Cloud allows either setup, local or remote.

EXAM TRAP

Remote servers that need OAuth do not work for cloud agent or code review when set in repository settings or custom agent profiles.

Serving a custom registry

A registry follows the MCP registry v0.1 spec and serves `GET /v0.1/servers`. Give Copilot the base URL only, because it appends the v0.1 path itself. The registry or its reverse proxy must send CORS headers on every `/v0.1/servers` endpoint. Once configured, Copilot CLI offers the registry for finding servers, and `/mcp search` queries it instead of the GitHub registry.

> **EXAM TRAP**
> Adding `/v0.1/servers` to the registry URL breaks it. Copilot adds that path automatically, so the doubled path errors out.

Registry access policies

The policy decides whether servers outside the registry may run. Nothing runs unless the MCP servers in Copilot policy is enabled for the enterprise or organization. A newly selected policy applies to all developers at once. Registry policies are in public preview and are not the recommended way to restrict servers.

Allow all and Registry only compared

Feature	Allow all	Registry only
Servers that can run	Any MCP server	Only servers in the registry
Local servers	No restriction	Must be in the registry with an exactly matching server ID
Copilot connectors	Usable	Not usable

Allowlists in managed settings

`managed-settings.json` **is the generally available, more secure control.** Its allowlist and denylist match servers by name, server URL or command. A malformed list counts as an empty `allowedMcpServers`, so every server except the built-in defaults is blocked. A server whose URL or command holds an unresolved variable such as `$VARIABLE` is blocked too.

> **EXAM TRAP**
> `allowedMcpServers` and `deniedMcpServers` cover servers from user and repository settings. They do not cover servers defined in repository custom agent profiles.

Key takeaways

- Copilot uses a configured server's tools without asking, so allowlist read-only tools rather than `*`.
- Only secrets prefixed `COPILOT_MCP_` reach the MCP configuration of Copilot cloud agent.
- Register only the base registry URL; Copilot appends `/v0.1/servers` itself.
- Registry only blocks connectors and unlisted local servers; Allow all has no restrictions.
- For the most secure restriction, use allowlists and denylists in `managed-settings.json`.

Sources: Configure MCP servers for your repository - GitHub Docs (opens in a new tab) · Adding MCP servers

for GitHub Copilot CLI - GitHub Docs (opens in a new tab) · Configuring an MCP server allowlist for your enterprise - GitHub Docs (opens in a new tab) · Configure an MCP registry for your organization or enterprise - GitHub Docs (opens in a new tab) · Model Context Protocol (MCP) servers, registries, and allow lists - Training | Microsoft Learn (opens in a new tab) · Restrict MCP server access to a custom registry - GitHub Docs (opens in a new tab)

Integrate agents within development environments

Start here

THINK FIRST

When an agent runs inside your repository or CI pipeline, what actually decides what it can see, change and trigger?

Why it matters: Questions describe where an agent runs and ask which repository, branch, token or environment setting limits what it can do.

An agent works inside an execution context: a repository, a target branch, a workflow and that workflow's permissions. In CI, the agent gets only the permissions that workflow grants. The context, not the agent's own file, enforces branch scope.

Key terms

Term	Meaning
Execution context	The repository, target branch, executing workflow and workflow permissions an agent runs under.
Branch-based scope	Limiting an agent to one branch, which the execution context enforces.
`GITHUB_TOKEN`	A short-lived, scoped token that GitHub Actions generates for each workflow run.
Cloud sandbox	The default place an agent session runs where users can choose; some entry points run only on GitHub Actions.

Read the execution context first

Four things frame every agent task: repo, branch, workflow, permissions. The agent works in one repository and targets one branch. A workflow runs the task, and the agent gets only the permissions granted to that workflow. Check these before judging what an agent can do.

EXAM TRAP

A custom agent file does not enforce branch scope. The execution context always does, whatever the agent profile says.

Scoped to one repository

An agent reads and changes code only in its own repository. It works with that repository's issues, pull requests and workflows. Other repositories stay out of reach unless access is explicitly granted. Chat sessions are scoped by default to one organization or to the user's personal repositories. Custom agents live in `.github/agents/` as files ending in `.agent.md`; `applyTo` narrows files and `tools` limits actions.

Invoking an agent in CI

In CI, the workflow becomes the agent's controlled execution path. The task gets a defined runner, trigger and repository, plus only the workflow's permissions. Copilot CLI run with `GITHUB_TOKEN` in an organization-owned repository bills usage to the organization. That needs an organization owner to enable the matching policy. After editing Agentic Workflows frontmatter, run `gh aw compile` before committing.

Example: Runs a named custom agent from a workflow step with a prompt and no user questions.

```
npx @github/copilot-cli --agent security-reviewer \
  -p "Review this code for vulnerabilities" --no-ask-user
```

- `--agent security-reviewer`: picks the custom agent to run
- `-p`: passes the prompt for the task

Branch-based scope

Agent changes go on a separate branch, not directly on main. That keeps every change isolated until review. When you delegate to Copilot coding agent, you can pick a base branch other than the default. Inside a CI workflow, the workflow decides the branch, and the agent runs in that branch's context.

Autonomous branches and pull requests

An agent can branch, edit, commit and open a pull request on its own. The flow runs: pick a base branch, create a working branch, change files, commit, then open the pull request. Copilot in the cloud stops earlier: it prepares changes for review but does not automatically create or merge a pull request. Cloud sessions default to interactive mode with assisted tool approvals.

EXAM TRAP

Allow-all mode is not always available. Users can turn it on only if their enterprise has not blocked it.

Tokens and workflow loops

Prefer `GITHUB_TOKEN` for automations in organization repositories. Every API request needs a token, such as a PAT, a GitHub App token or `GITHUB_TOKEN`. GitHub Agentic Workflows add two guards: tokens are read-only by default, and safe outputs let the agent propose actions that a separate gated step approves.

How the token choice affects extra workflow runs

Feature	GITHUB_TOKEN	PAT or GitHub App token
Extra workflow runs	Most actions trigger none	Can trigger them, depending on configuration
Loop risk	Helps prevent unintended loops	Needs careful design to avoid recursive runs
Stored credentials	None; a new token per run	Not covered here

Cloud sandbox or GitHub Actions

Sessions run in a cloud sandbox by default, or on GitHub Actions. Both draw compute from the same monthly allowance. Some entry points run only on Actions, such as mentioning `@Copilot` in a pull request or giving Copilot an issue. Organization owners can set a firewall that limits which external hosts the cloud agent reaches.

Choosing where an agent session runs

Feature	Cloud sandbox	GitHub Actions
Default where users can choose	Yes	No
Environment control	Not described	Secrets, variables and setup steps for dependencies
Runners and networks	Not described	Self-hosted runners or private networking

Environment-specific constraints

Environments add approvals, secret limits and target control for deployments. Each job can reference a single environment. Its protection rules must pass before the job goes to a runner, and secrets unlock only after that. Repository and organization settings also limit which workflows and actions run and what `GITHUB_TOKEN` may do.

> **EXAM TRAP**
> Triggers matter too, so review them carefully. Workflows that run on pull request events from forks are a higher-risk entry point for an agent.

Key takeaways

- The execution context, not the custom agent, enforces an agent's branch-based scope.
- Agents stay in their own repository unless access to another one is explicitly granted.
- In CI, the workflow picks the branch, runner, trigger and permissions for the agent.
- `GITHUB_TOKEN` rarely triggers extra runs; PATs and App tokens can cause loops.
- Environment protection rules must pass before a job reaches a runner or its secrets.

Sources: Execution context and boundaries - Training | Microsoft Learn (opens in a new tab) · Copilot CLI in

GitHub Actions - GitHub Docs (opens in a new tab) · Managing GitHub Copilot in the cloud for your enterprise or organization - GitHub Docs (opens in a new tab) · Develop agentic workflows in GitHub Actions - GitHub Docs (opens in a new tab) · Agent execution limits and protections - Training | Microsoft Learn (opens in a new tab) · How agents interact with GitHub APIs and workflows - Training | Microsoft Learn (opens in a new tab) · Deployment environments - GitHub Docs (opens in a new tab)

Operate agents with safe execution paths and robust error handling

Start here

THINK FIRST

When an agent hits an error mid-task, how should it retry, stop or involve a person, and how do you trace what it did?

Why it matters: Questions describe an agent error or risky action and ask which hook, retry, escalation path or audit record handles it.

Hooks let you intercept errors and tool calls, so you can retry, skip or abort, and approve or deny risky actions. When the firewall blocks a request, a visible warning reports it. Signed commits with linked session logs make every agent change traceable.

Key terms

Term	Meaning
`onErrorOccurred`	A Copilot SDK hook that is called when an error occurs during session execution.
`errorHandling`	A hook output field that sets the response to an error: `retry`, `skip` or `abort`.
`preToolUse`	A hook that runs before the agent uses any tool and can approve or deny that use.
Co-author	The developer who assigned the issue or asked for the change, credited on Copilot's commits.

Catch errors with hooks

`onErrorOccurred` **runs when a session hits an error.** Its `errorContext` input says where the error arose: `model_call`, `tool_execution`, `system` or `user_input`. One use is triggering alerts for critical errors. Returning null or undefined keeps the default handling. The agent hook event `errorOccurred` can also log errors, send notifications and track error patterns.

EXAM TRAP

The Java SDK has no `onErrorOccurred` hook. It uses `EventErrorPolicy` and `EventErrorHandler` instead.

Choose retry, skip or abort

The hook's output fields decide what happens next. Set `errorHandling` to pick the response, and `retryCount` to cap how many retries run. Use `suppressOutput` only when the user should not see the error output.

Error hook output fields and what they control

Output field	Value	Effect
`errorHandling`	`retry`, `skip` or `abort`	How the error is handled
`retryCount`	A number	How many retries run when `errorHandling` is `retry`
`suppressOutput`	`true`	The error output is hidden from the user

Retry a stuck agent

Stuck cloud agent work has simple manual retries. If a session on GitHub Actions stays stuck, unassign the issue and assign it to Copilot again. If Copilot stalls on a pull request comment, post that comment again. To watch progress, open View session on the "Copilot started work" event; its logs stream live.

EXAM TRAP

Commenting on a merged or closed pull request will not restart Copilot. It responds to mentions only while the pull request is open.

Gate risky actions

`preToolUse` **can approve or deny any tool call.** Use it to block dangerous commands or require approval for sensitive operations. `postToolUse` runs after every tool, successful or failed, and can send failure alerts. In Copilot CLI, you can reject a tool permission request with inline feedback. Copilot cloud agent reads hooks only from the default branch.

Escalate blocked work to people

Blocked requests surface as visible warnings. When the firewall blocks a request, a warning naming the address and command appears in the pull request body or a comment. The firewall limits internet access by default to manage data exfiltration risk. A ruleset or branch protection rule can block Copilot's pull requests. Repository administrators can set a ruleset bypass for Copilot.

Validate before pushing

Copilot can test its work in an ephemeral environment. There it runs automated tests and linters before it pushes. It does this most reliably with clear instructions, best kept in `.github/copilot-instructions.md`. After pushing, Agent Merge, turned on with Watch and fix, works on failed workflows, merge conflicts, or both.

> **EXAM TRAP**
> An attached image over **3.00** MiB never reaches Copilot cloud agent. It is removed from the request.

Make agent actions traceable

Every agent commit links to its session logs. That link shows reviewers and auditors why a change was made. Commits are authored by Copilot, with the requesting developer as co-author, and signed so GitHub shows them as "Verified". Administrators can see session logs and audit log events. Chat data for sessions on GitHub Actions is kept for **28** days.

Key takeaways

- The `errorHandling` output of `onErrorOccurred` can be `retry`, `skip` or `abort`.
- `retryCount` sets how many retries run when `errorHandling` is `retry`.
- `preToolUse` can deny a tool call; `postToolUse` runs after success or failure.
- Retry a stuck session by unassigning the issue and assigning Copilot again.
- Copilot's signed commits link to session logs and credit the requester as co-author.

Sources: Error handling hook - GitHub Docs (opens in a new tab) · Hooks for GitHub Copilot - GitHub Docs (opens in a new tab) · Troubleshooting GitHub Copilot cloud agent - GitHub Docs (opens in a new tab) · Risks and mitigations for GitHub Copilot cloud agent - GitHub Docs (opens in a new tab) · Managing agent sessions - GitHub Docs (opens in a new tab) · Steering agents in GitHub Copilot CLI - GitHub Docs (opens in a new tab) · Customize agent workflows with hooks - GitHub Docs (opens in a new tab) · GitHub Copilot on GitHub.com and GitHub Mobile - GitHub Docs (opens in a new tab)

Domain 3 | 10–15% of the exam | 13 min read

Manage memory, state, and execution

Overview

Every point below comes from GitHub's official documentation; each section lists its sources.

Implement agent memory strategies

Start here

THINK FIRST

How do you decide what an agent should remember, for how long, and when to clear it?

Why it matters: Questions describe an agent's memory needs and ask which memory layer, scope, or expiry rule fits.

Agent memory works in layers. The conversation context holds the current task and lasts only for that session. Copilot Memory keeps lasting facts and preferences, which expire after a default period that restarts when Copilot upvotes them.

Key terms

Term	Meaning
Conversation context	What the current session already holds; a new session starts without it.
Repository-level fact	A memory that describes a codebase and is shared with your collaborators.
User-level preference	A memory about how you like to work that follows you from repository to repository.
Expiry period	How long a memory lives; it starts at creation and restarts when Copilot upvotes it.
Subagent	A helper agent with its own separate context window, focused on one assigned task.

Repository facts vs user preferences

Copilot Memory has two scopes: the repository and the user. Repository facts describe the codebase for everyone working in it. User preferences describe your own habits. When you ask for a memory, state its scope plainly instead of relying on words like "always".

The two Copilot Memory scopes compared

Feature	Repository-level fact	User-level preference
Describes	The codebase	How you personally like to work
Who benefits	Your collaborators in that repository	You personally
Across repositories	Not pooled across an organization, and not copied to a fork	Follows you between repositories

EXAM TRAP

Scope does not depend on who stated the memory. Copilot picks repository or user scope based on who the memory applies to.

Store only lasting facts

Memory is for things that stay true over time. Copilot is told to skip short-lived or task-specific instructions. Use custom instructions for important rules, and let memory collect the growing detail that would bloat an instructions file. If a matching memory exists, Copilot can reinforce it rather than add a duplicate.

EXAM TRAP

Asking Copilot to remember something does not guarantee it is saved. The model weighs the prompt and the context before storing anything.

Keep each context focused

Load only what the current task needs. A subagent works in its own context window, so its intermediate steps stay out of the main conversation. Agent skills load a workflow's resources only when the agent works on that task. In Copilot CLI, subagents with `read_memories` access can read memories. Only the root agent can store or vote on them.

Guardrails on new memories

Copilot CLI can ask you to approve a memory update. You can approve one update, reject it with optional feedback, or allow future updates for the current repository, directory, or session. Any memory containing a secret, such as a token or key, is rejected. On GitHub.com you cannot edit a single memory in settings; you ask Copilot to add it.

Turning memory off and on

Turning memory off does not clear the current context. It stops new storage and retrieval, but memories already given to the conversation can stay. Start a new session to drop them. In non-interactive runs, memory is off by default, even when it is on for interactive sessions.

Example: Runs a single non-interactive prompt with Copilot Memory turned on for that run.

```
copilot -p "YOUR PROMPT" --enable-memory
```

- `--enable-memory`: enables memory for this run only

Importing prepared memories

Copilot CLI can load memories from a JSONL file. Each line holds one versioned memory object. Required fields are type, version, scope, subject, fact, reason, and citations; externalId is optional. Running the import counts as approval, but secret scanning still applies. A conflict errors by default, unless you add `--on-conflict skip`.

EXAM TRAP

Rerunning the whole import after a partial failure can create duplicates. Resubmit only the records that failed.

Key takeaways

- Memory should hold facts that stay true over time, not short-lived or task-specific instructions.
- Repository facts are shared with collaborators; user preferences follow you between repositories.
- Default expiry is **90 days**; owners can choose **7** to **180** days, and the lower setting wins.
- To drop memories already in context, turn memory off and start a new session.

Sources: Managing Copilot Memory for your personal account - GitHub Docs (opens in a new tab) · About GitHub Copilot Memory - GitHub Docs (opens in a new tab) · Custom agents - GitHub Docs (opens in a new tab) · Manually creating memories for GitHub Copilot - GitHub Docs (opens in a new tab) · Agent skills - GitHub Docs (opens in a new tab)

Persist agent state and manage context drift

Start here

THINK FIRST

How do you keep a long-running agent on track and pick up its work later without losing earlier decisions?

Why it matters: Questions describe a long or resumed agent task and ask how to save its state, resume it, or spot and fix drift.

Long sessions outgrow the context window, so Copilot compacts old history into a summary that keeps decisions and next steps. Checkpoints save those summaries, and workflow artifacts keep job data after the job ends. When the agent drifts, you can steer it mid-run.

Key terms

Term	Meaning
Compaction	Replacing old conversation history with a structured summary plus the original instructions and plan state.

Term	Meaning
Checkpoint	A saved compaction summary; in Copilot CLI, a numbered file with a title.
Workflow artifact	Data a job saves so it persists after the job ends and another job in the workflow can use it.
Steering	Giving a running agent new input mid-session without stopping the run.
Cloud sandbox session	A Copilot session whose files and in-progress work live in GitHub-hosted infrastructure, not on your device.

How compaction keeps continuity

Compaction swaps old history for a structured summary. The summary is kept with the user's original instructions and the latest state of any plans or to-do lists. With it, Copilot still knows what was discussed, what was decided and what comes next, even though the original messages are gone.

When compaction runs

Copilot CLI compacts in the background at about 80% context use. That leaves about **20%** of the window so tool calls keep running. If usage reaches about **95%** first, the CLI pauses briefly until compaction finishes. You can also run `/compact` at any time in Copilot CLI and the GitHub Copilot app. Xcode does not support compaction.

EXAM TRAP

Wrong: compaction waits until the context window is full. In Copilot CLI it starts automatically at about **80%**, keeping a buffer for tool calls.

Checkpoints as saved progress

A checkpoint is a saved compaction summary you can inspect. In Copilot CLI, checkpoints are numbered files with titles. If Copilot seems to forget a decision or contradicts earlier work, open the checkpoints. They show what compaction kept and what it may have summarized differently than you expected.

EXAM TRAP

Wrong: a compaction summary keeps every detail. Exact message wording, full command output and minor decisions from early in a long conversation may be left out.

Recording decisions with rationale

A rationale and confidence level turn an agent decision into a record. In GitHub Agentic Workflows, set `issue-intent: true` on an agent output to require both for that output. If the agent leaves them out, the workflow fails. An automation's agent adds both on its own whenever it changes a supported issue attribute.

Workflow artifacts for durable data

Artifacts keep job data after the job completes. Another job in the same workflow can then use that data. Name each artifact; otherwise it gets the default name `artifact`. On download, `download-artifact` checks the artifact digest against the upload step's output. A custom `retention-days` value cannot exceed the repository, organization or enterprise limit.

EXAM TRAP

Wrong: overwrite an artifact by re-uploading under the same name. Artifacts are immutable in upload-artifact v4, so an updated file needs a different artifact name.

Resuming a session

Resume a session to continue with its existing context. This avoids starting over. In Copilot CLI, use the Sessions tab or type `/resume`; cloud sessions appear next to local ones. Cloud sandbox sessions are not tied to the device that started them. Resuming one in the CLI brings back its full history and work.

EXAM TRAP

Wrong: session querying always works. Session data syncs by default, but with `"remoteExport": false` set in CLI settings, session querying is not available.

Catching drift during a run

Watch the live session log and steer as soon as the agent drifts. From the Agents tab, open any agent session to follow its progress and thought process in real time. If the request was scoped wrong, send steering input without stopping the run. Each steering message consumes AI credits.

Reset options when drift persists

Pick how much context to keep: all, some or none. Resume keeps everything. Clearing context keeps the session but drops the conversation. A fresh session suits work that went the wrong way.

Ways to continue or reset an agent's context

Option	What it keeps	Best when
Resume a session	The existing conversation context	You want to continue without starting over
Clear context	Session identity, system and developer messages, configuration, event log	You want the same session without the model-facing conversation
Start a fresh session	Nothing from the earlier conversation	Work went wrong and you want a clean slate, not reconciled decisions

EXAM TRAP

Wrong: turning off Copilot Memory clears it from the current chat. Memories already supplied can stay in context, so start a new session to work without them.

Key takeaways

- Compaction keeps a summary, the original instructions and plan state, but minor early decisions can be lost.
- If Copilot contradicts earlier work, check the checkpoints to see what compaction kept.
- Artifacts are immutable in upload-artifact v4, so upload a changed file under a new name.
- Resume a session to keep its context; start fresh when the work went the wrong way.
- Steer a running agent without stopping it; each steering message consumes AI credits.

Sources: Managing context for GitHub Copilot - GitHub Docs (opens in a new tab) · Workflow artifacts - GitHub Docs (opens in a new tab) · Agent management - GitHub Docs (opens in a new tab) · Managing agent sessions - GitHub Docs (opens in a new tab) · Managing rationale, confidence, and approvals for issues - GitHub Docs (opens in a new tab) · Store and share data with workflow artifacts - GitHub Docs (opens in a new tab) · Context clearing and terminal tools - GitHub Docs (opens in a new tab) · Managing Copilot Memory for your personal account - GitHub Docs (opens in a new tab)

Ensure continuity of agent memory and state across tools and environments

Start here

THINK FIRST

How do you keep an agent's memory shared, consistent and current as work moves between tools and sessions?

Why it matters: Questions describe agents sharing memory or context and ask how to share state, avoid conflicts or avoid stale facts.

Copilot Memory keeps facts that stay true over time and shares them across Copilot surfaces. Scope and precedence rules keep context from conflicting. Citation checks, votes and expiry keep it from going stale.

Key terms

Term	Meaning
Repository-level fact	A memory that describes a codebase and is shared with your collaborators on it.
User-level preference	A memory that applies to you and follows you from one repository to another.
Expiry period	How long a memory lives: **90** days by default, reset when Copilot upvotes it.
Compaction	Copilot replacing old conversation history with a summary to free room in the context window.
Checkpoint	What you check when Copilot seems to forget a decision, to see what compaction preserved and what it summarized differently.

Share state across surfaces

A memory captured on one Copilot surface serves the others. If Copilot cloud agent works out your repository's approach to database connections, Copilot code review can apply it later. Repository facts are shared with your collaborators. They are not pooled across an organization's repositories, though.

EXAM TRAP

Forking a repository does not bring its memories along. The fork starts without the original's stored repository facts.

Choose the right scope

Scope follows who a memory applies to, not who said it. Words like "always" or "going forward" do not set scope, so say it explicitly. When your intent is unclear, Copilot asks you to clarify rather than guess.

Repository facts and user preferences compared

Feature	Repository-level fact	User-level preference
Applies to	Everyone working in the repository	You, across repositories
Delete it immediately in	Repository settings	Personal settings

Who can change memory

In Copilot CLI, only the root agent writes memories. Only the root agent can call `store_memory` or `vote_memory`. Subagents with `read_memories` access can read memories but never change them. Subagents also get their own context windows, which keeps the main agent's context focused.

Settle conflicting instructions

Precedence decides which instructions win. For Copilot Chat on GitHub.com, personal instructions rank highest and organization custom instructions lowest. Path-specific instruction files outrank `.github/copilot-instructions.md`, which outranks `AGENTS.md`. In pull request review, Copilot reads instructions and skills from the head branch, not the base branch.

EXAM TRAP

Asking Copilot to store a memory does not guarantee it is stored or always used. Put rules that must apply every time in custom instructions.

Keep memories fresh

Copilot checks a fact's citations against the current branch. That check tells Copilot whether a repository fact is still accurate before it relies on it. It upvotes facts it can validate and downvotes wrong or outdated ones. Copilot is instructed not to store short-lived, task-specific instructions. When a matching memory exists, Copilot can reinforce it instead of adding a duplicate.

How memories expire

Memories expire after 90 days by default. Owners can set repository-fact expiry anywhere from **7** to **180** days. If an organization and its enterprise set different values, the lower one applies. An upvote resets the period, but simply retrieving a memory does not.

EXAM TRAP

Telling Copilot to "forget" a memory only downvotes it. To delete it right away, use your personal or repository settings.

Stale context in long sessions

Compaction summarizes old history to free the context window. Copilot CLI starts compacting in the background at about **80%** of the window. The summary keeps your original instructions and current plans, but fine detail can be lost. A finished compaction cannot be undone. If Copilot contradicts earlier work, check its checkpoints to see what compaction preserved.

Start clean when needed

A new session clears old context. Start fresh when work went in a wrong direction, so Copilot stops reconciling old decisions with the new approach. Turning off memory in Copilot CLI stops storage and retrieval. Memories already loaded stay in the current conversation until you start a new session.

Key takeaways

- Repository facts are shared with collaborators, but not across repositories or into forks.
- Only the root agent in Copilot CLI can store or vote on memories.
- Memories expire after **90** days by default; upvotes reset the clock, reads do not.
- In non-interactive runs such as `copilot -p`, memory is off unless you add `--enable-memory`.
- Use custom instructions, not memory, for rules that must always apply.

Sources: About GitHub Copilot Memory - GitHub Docs (opens in a new tab) · Manually creating memories for GitHub Copilot - GitHub Docs (opens in a new tab) · Custom instructions for GitHub Copilot - GitHub Docs (opens in a new tab) · Managing context for GitHub Copilot - GitHub Docs (opens in a new tab) · Managing Copilot Memory for your personal account - GitHub Docs (opens in a new tab)

Domain 4 | 15–20% of the exam | 15 min read

Perform evaluation, error analysis, and tuning

Overview

Every point below comes from GitHub's official documentation; each section lists its sources.

Define success criteria and evaluation signals for agent tasks

Start here

THINK FIRST

Before an agent starts, how do you decide what a successful result looks like and how you will measure it?

Why it matters: Questions describe an agent task and ask which outcomes, constraints and evaluation signals show the work truly succeeded.

Define each agent task by its inputs, its outputs and its success criteria. The criteria should match the task's real intent, not just a green build. Signals from checks, scans and reviews then tell you whether the result meets them.

Key terms

Term	Meaning
Success criteria	How an agent's results are judged: through checks, scans and review outcomes.
Operational constraint	A boundary on the task, such as adding no secrets or never pushing directly to main.
Resolution rate	The percentage of tasks an agent completes successfully; a key evaluation metric.
Required check	A CI workflow a pull request must pass before it can merge, set through rulesets or branch protection.
Hook	Code that runs at key points of agent work and enforces a rule instead of asking the model.

Specify the expected outcome

Describe the outcome you want and its constraints. A good issue gives a clear problem, complete acceptance criteria and hints about which files to change. Vague tasks invite changes that look plausible but miss the real problem. The Optional prompt field can add coding patterns, files to modify or testing requirements.

> **EXAM TRAP**
> Comments added to an issue after assignment never reach Copilot. Post follow-up details on the pull request instead.

Set operational constraints

Constraints state what the agent must not do. Typical ones: workflow edits need platform review, no secrets may be added, and nothing goes straight to main. Another criterion is that scope matches intent, with no unexpected files changed. Hooks enforce rules by running code, and `preToolUse` can approve or deny any tool call.

Align criteria with intent

Passing CI is necessary but not always sufficient. Write criteria around the real goal, such as "vulnerability resolved" rather than "tests passed". Making the CI workflow a required check blocks merging until it passes. Copilot can still produce incorrect or insecure code, so review and test it before production.

Plan before code

Plan mode settles requirements before any code changes. The plan agent researches the task with read-only tools and codebase analysis to find its requirements and constraints. Its plan gives a summary, a breakdown of steps and open questions. You can iterate on it to clarify requirements or adjust scope.

Quantitative signals

Measure resolution rate, token efficiency, latency and tool call reliability. GitHub uses these metrics to evaluate Copilot. Each evaluation includes several independent runs, because model outputs are nondeterministic. For one session, the session log and overview show progress, token usage and session length.

Qualitative signals

Review comments carry a High, Medium or Low severity. Attributions under each comment name the agent skill or MCP server that produced it. Rating comments helps improve future suggestions.

Evaluation signals and what each tells you

Signal	Kind	What it tells you
Resolution rate	Quantitative	Share of tasks completed successfully
Session token usage and length	Quantitative	How much one session used and how long it ran

Signal	Kind	What it tells you
Review comment severity	Qualitative	How important each issue found is
Session history query	Qualitative	Sessions where the agent struggled or needed repeated correction

EXAM TRAP

A default Copilot code review leaves a Comment review. It does not count toward the approvals a pull request requires.

Signals from automated scanning

Copilot cloud agent scans its own code by default. While generating, it runs CodeQL, secret scanning and dependency analysis on new code. New dependencies are checked in the GitHub Advisory Database for malware and High or Critical vulnerabilities. This needs no GitHub Advanced Security license. The agent also gets a second opinion from Copilot code review.

EXAM TRAP

Copilot does not re-review new pushes on its own. Configure automatic reviews to include new pushes if you want that.

Checks inside agent work

Hooks can run lint, format and security checks during execution. Copilot cloud agent also runs tests and linters in its own ephemeral environment before pushing. Test a hook locally by piping input into it, checking its exit code and validating its JSON output. Code scanning alerts in a security campaign can be handed to Copilot to fix.

Key takeaways

- Define every agent task by its inputs, outputs and success criteria.
- Write criteria for the real intent; passing CI alone may not be enough.
- Track resolution rate, token efficiency, latency and tool call reliability.
- Copilot cloud agent scans new code with CodeQL, secret scanning and dependency analysis.
- Default Copilot reviews are Comment reviews and do not count as approvals.

Sources: Define inputs, outputs, and success criteria - Training | Microsoft Learn (opens in a new tab) · Application card: GitHub Copilot Agents - GitHub Docs (opens in a new tab) · Hooks for GitHub Copilot - GitHub Docs (opens in a new tab) · Customize agent workflows with hooks - GitHub Docs (opens in a new tab) · Kick off a task with Copilot agents on GitHub - GitHub Docs (opens in a new tab) · GitHub Copilot on GitHub.com and GitHub Mobile - GitHub Docs (opens in a new tab) · Using plan mode in your IDE - GitHub Docs (opens in a new tab) · Managing agent sessions - GitHub Docs (opens in a new tab)

Analyze agent failures and identify root causes

Start here

THINK FIRST

When an agent's workflow run fails, how do you tell a reasoning error from tool misuse or an environment problem?

Why it matters: Questions describe a failed agent run and ask where to find the evidence and which kind of root cause it points to.

GitHub makes an agent's actions traceable through workflow logs, commit history and pull request discussions. During a session, you can also follow a session log that shows what the agent is doing. The steps below show where to look in that evidence.

Key terms

Term	Meaning
Workflow artifact	Files a job saves, such as build logs, kept after the run ends and shareable with other jobs.
Check run	The record GitHub creates for each job inside the check suite of a workflow run.
Expanded line	The part of an expression log that shows the `if` condition with its runtime values filled in.
Transient failure	A temporary failure, such as a network issue, that a retry is meant to handle.
Execution context	The setting an agent runs in. It enforces branch-based scope, which the custom agent itself does not set.

Read the run logs first

A failed run's logs are where diagnosis starts. You can view and search the build logs of a failed run, then re-run it. Search only looks inside expanded steps, so expand a step before you search it. Each job also logs two steps GitHub adds itself: "Set up job" and "Complete job".

EXAM TRAP

A log search that finds nothing does not prove the text is absent. Collapsed steps are left out of the search results.

Narrow the log to failures

`--log-failed` **shows only the failed steps.** In GitHub CLI, swap `--log` for `--log-failed` to cut the output down to the steps that failed. Each run has one check suite with a check run per job. Copilot CLI includes the GitHub MCP server, so it can read runs, job logs and check statuses directly.

Where failure evidence lives

Artifacts keep logs and files after the run ends. Use artifacts, not caching, when you need build logs or binaries after a run. By default, logs and artifacts are kept for **90** days. You can customize that period, depending on the repository type.

Evidence sources for a failed run and their limits

Evidence	What it shows	Watch for
Build logs	Step output of a failed run, searchable	Search skips collapsed steps
Log archive after a partial re-run	Logs of the jobs you re-ran	Jobs you did not re-run are missing
Artifacts	Saved files such as binaries or build logs	Deleted together with the workflow run
Expression log	How a job-level `if` was evaluated	Not written for step-level conditions

EXAM TRAP

Deleting a workflow run also deletes its artifacts from storage. Download the evidence before you clean up old runs.

Why a job ran or was skipped

Job-level `if` evaluations are logged. GitHub Actions evaluates a job's `if` itself and sends the job to a runner only when it is true. The log explains a skipped job and one that ran unexpectedly. Find it in `JOB-NAME/system.txt` inside the downloaded log archive. Its Expanded line shows the actual runtime values.

EXAM TRAP

Expression logs exist only for job-level conditions. For a step-level `if`, turn on debug logging to see the evaluation in the job logs.

Environment and timing causes

Some failures come from the environment, not the code. A test that passes locally but fails in CI with a one-day difference may be a timezone issue, because the runner uses UTC. Comparing a passing and a failing log can expose a race condition, like a test not waiting for a background job. GitHub Actions contexts hold run values, and most can be used at any point in a workflow. Contexts change with the run's conditions, so a context can be empty in some jobs.

Tool and access limits

An agent can only act within its tools and access. A custom agent's `tools` field defines its actions, and `applyTo` limits the files it focuses on. Copilot in the cloud reaches only the repository it works in. Agentic Workflows give the agent read-only tokens by default. Check these limits before you blame the agent's reasoning.

EXAM TRAP

Branch-based scope is not set by a custom agent's `tools` or `applyTo`. The execution context the agent runs in always enforces it.

When a retry helps

Retries are for temporary failures. Use a retry for network issues or other transient errors. A failure on every new commit has a different cause. An invalid workflow file in `.github/workflows` produces a failed run for each new commit, so fix the file instead.

Key takeaways

- Expand steps before searching a failed run's logs; search skips collapsed steps.
- Use `--log-failed` in GitHub CLI to see only the failed steps.
- Logs and artifacts stay **90** days by default, and deleting a run deletes its artifacts.
- Job-level `if` evaluations are in `system.txt`; step-level ones need debug logging.
- A local pass with a CI failure can come from the runner using UTC.

Sources: Agent execution limits and protections - Training | Microsoft Learn (opens in a new tab) · How agents interact with GitHub APIs and workflows - Training | Microsoft Learn (opens in a new tab) · Workflow artifacts - GitHub Docs (opens in a new tab) · Using workflow run logs - GitHub Docs (opens in a new tab) · Viewing job condition expression logs - GitHub Docs (opens in a new tab) · Execution context and boundaries - Training | Microsoft Learn (opens in a new tab) · Diagnosing CI test failures - GitHub Docs (opens in a new tab) · Downloading workflow artifacts - GitHub Docs (opens in a new tab)

Tune agent behavior based on evaluation results

Start here

THINK FIRST

When an evaluation shows an agent behaving badly, should you change its instructions, its memory, or its tools?

Why it matters: Questions describe an evaluation finding and ask which instruction, workflow, memory, or tool setting to change.

Tuning an agent means adjusting three levers: the instructions and workflows it follows, the memory it draws on, and the tools it may use. Rules that must hold every time belong in custom instructions, because a stored memory is not guaranteed to be used.

Key terms

Term	Meaning
Custom instructions	Markdown files, such as `.github/copilot-instructions.md`, that tell Copilot how to work in a repository.

Term	Meaning
Copilot Memory	Stored repository-level facts and user-level preferences that Copilot can retrieve in later work.
Safe outputs	Validated write actions declared in agentic workflow frontmatter; the only way the agent may write.
MCP server	A server that gives an agent extra tools, such as the GitHub or Playwright MCP server.
Tool search	A Copilot CLI mode that avoids loading every tool at once; it is skipped below about **30** tools.

Where instructions live

Each instruction file type has its own location and reach. Put rules for every task in the root `.github/copilot-instructions.md` file. Those rules also reach Copilot Chat and Copilot code review. Narrow a rule to certain file types with a glob in the front matter of a path-specific file.

Instruction file types compared

Instruction type	File location	What it covers
Repository-wide	`.github/copilot-instructions.md` in the repository root	All tasks given to Copilot, plus Chat and code review
Path-specific	`.github/instructions/**/*.instructions.md`	File types matched by the glob in its front matter
Personal	`~/.copilot/copilot-instructions.md` or `~/.copilot/instructions/**/*.instructions.md`	Your own personal instructions
Agent instructions	`AGENTS.md`, `CLAUDE.md`, `GEMINI.md` or `REVIEW.md`	Supported by Copilot code review

EXAM TRAP

Storing a must-follow rule as a memory is the wrong fix. A saved memory may still go unused, so consistent rules belong in custom instructions.

Revising an agentic workflow

Edit the markdown, recompile, then push both files. Change the workflow file in `.github/workflows/` and run `gh aw compile` to refresh its lock file. Commit both files and check the GitHub Actions results in a pull request. Use `gh aw logs` and `gh aw audit` to compare duration, tokens, and cost.

Example: Reviews recent runs, inspects one run, then rebuilds the lock file after an edit.

```
gh aw logs
gh aw audit RUN-ID
gh aw compile
```

- `gh aw audit RUN-ID`: shows token usage and estimated inference cost for one run

Constraints in workflow frontmatter

Frontmatter limits what the agent can read, write, and spend. The `permissions` field defaults to read-all. Writes such as new issues or pull requests happen only through declared safe outputs. Secrets stay in isolated downstream jobs. The `engine` field picks the coding agent, with GitHub Copilot as the default.

Example: Keeps the agent read-only and lowers its per-run inference cap.

```
permissions: read-all
max-ai-credits: 500
```

- `max-ai-credits: 500`: caps inference usage for a single run

EXAM TRAP

Leaving out `max-ai-credits` does not mean unlimited spend. Each run still has a default cap of **1,000** AIC.

Autonomy and tool approvals

Give risky tasks less autonomy instead of one rule for all. Cloud sessions start in interactive mode with assisted tool approvals. Users can turn on allow-all mode unless the enterprise blocks it. Enterprises use `managed-settings.json` to disable allow-all or assisted approval, or to set permissions per operation.

Memory scope and lifetime

By default, memories expire 90 days after creation or last upvote. Copilot decides if a memory is a user-level preference or a repository-level fact by whom it applies to, not who said it. So say the scope plainly. Owners can set fact retention from **7** to **180** days, and the shorter of organization and enterprise settings applies.

EXAM TRAP

Retrieving a memory does not extend its life. Only its creation or an upvote starts the default **90**-day clock again.

Turning memory off and cleaning up

Turning memory off does not erase what is already stored. Pick the action that actually removes unwanted context. Users with write access can review stored repository facts in settings and delete wrong or misleading ones.

Memory actions and their limits

Action	Effect	Limit or next step
Turn off memory in the CLI	Stops further storage and retrieval	Memories already in the conversation stay; start a new session
Disable memory use	Copilot stops using memories	Stored memories are not deleted
Ask Copilot to forget	Downvotes the memory	To delete now, use personal or repository settings
Disable memory for a repository	Repository memory stops	User-level preferences still apply; disable at organization or enterprise level
Run `copilot -p`	Memory is off by default for that run	Add `--enable-memory` to turn it on

EXAM TRAP

Telling Copilot to forget something does not delete it. Copilot only downvotes the memory, which stays in the store.

Controlling tool access

Custom agents inherit repository MCP tools unless you restrict them. You can limit a custom agent to specific tools. The GitHub and Playwright MCP servers are on by default for cloud agent. Organization owners use the firewall to choose which external hosts and URLs the agent can reach.

EXAM TRAP

A `deniedMcpServers` entry in `managed-settings.json` will not block a server defined in a repository custom agent profile. Those settings cover user and repository settings only.

Tuning how tools load

Tool search pays off only above roughly 30 tools. With fewer tools, Copilot CLI simply loads them all. A Markdown custom agent that names many tools can add `deferred-tool-loading: true` to its frontmatter. Set an MCP server's `deferTools` to `"never"` to keep its tools always listed.

Key takeaways

- Put rules that must always apply in custom instructions, not memory, since a stored memory may go unused.
- After editing an agentic workflow, run `gh aw compile` and commit both the markdown and the lock file.
- Asking Copilot to forget only downvotes a memory; delete it in personal or repository settings.
- Subagents in Copilot CLI can use memories, but only the root agent may store or vote on them.
- Custom agents inherit repository MCP tools by default, but you can limit them to specific tools.

Sources: Best practices for using GitHub Copilot to work on tasks - GitHub Docs (opens in a new tab) · Manually creating memories for GitHub Copilot - GitHub Docs (opens in a new tab) · Managing Copilot Memory for an enterprise, organization, or repository - GitHub Docs (opens in a new tab) · GitHub Agentic

Workflows - GitHub Docs (opens in a new tab) · Loading tools on demand with tool search - GitHub Docs (opens in a new tab) · Support for different types of custom instructions - GitHub Docs (opens in a new tab) · Creating GitHub Agentic Workflows - GitHub Docs (opens in a new tab) · Build reliable workflows - outputs, contexts, triggers, and cross-job handoffs - Training | Microsoft Learn (opens in a new tab)

Domain 5 | 15–20% of the exam | 18 min read

Orchestrate multi-agent coordination

Overview

Every point below comes from GitHub's official documentation; each section lists its sources.

Operate and manage multi-agent workflows

Start here

THINK FIRST

When you split one task across several agents, how do you coordinate them and keep their work from clashing?

Why it matters: Questions describe a multi-agent task and ask how to orchestrate it, isolate parallel agents or resolve conflicts between them.

An orchestrator, usually the main agent, tracks the overall goal and hands work to subagents. Each subagent works in its own context window, often in parallel. At the end, the results are reviewed and combined.

Key terms

Term	Meaning
Orchestrator	The agent, usually the main one, that tracks the overall goal and delegates work to subagents.
Subagent	An agent doing delegated work in its own separate context window.
Fleet mode	The `/fleet` command, where the main agent splits a prompt into subtasks for subagents.
Workspace isolation	Running app sessions in separate git worktrees or cloud sandboxes so concurrent tasks stay apart.
Dynamic workflow	An experimental Copilot CLI workflow made of steps and subagents, whose saved results a resumed run can reuse.

Orchestrate with /fleet

The main agent decides whether a prompt can be split. With `/fleet`, the main agent looks for smaller subtasks in the prompt. If it delegates, it becomes the orchestrator and manages dependencies between subtasks. Where possible, it runs subagents in parallel to finish sooner. The `/plan` command can shape the breakdown first.

EXAM TRAP

More agents do not always help. A sequential request may gain nothing from `/fleet`, and separate model calls per subagent can use more AI Credits.

When parallel work pays off

Split work only when tasks are independent with clear boundaries. Coordinating the results must cost less than the time it saves. A subagent works in a context window of its own, so it does not carry all of the main session's context. In Copilot CLI, custom subagents do not get repository instructions by default.

Isolate agents running in parallel

Give each concurrent task its own workspace. GitHub Copilot app sessions can each use a separate git worktree or cloud sandbox, apart from the default branch. Every cloud agent session gets a temporary environment that is destroyed when it ends, so no state carries over. Each SDK session has its own context, tools and permissions, and MCP servers run as separate processes.

Scope each agent's tools

A custom agent's profile settings limit what it can do. Add the `agent` alias to the tools list to let one custom agent invoke another. For Copilot in the cloud, the built-in `github` MCP server offers read-only tools by default, with a token scoped to the originating repository.

Custom agent profile settings and their effects

Setting	Value	Effect
`tools`	omitted	Every available tool is enabled
`tools`	`[]`	Every tool is disabled
`tools`	a specific list	Only the listed tools are enabled
`user-invocable`	`false`	Not selectable by users; reached only programmatically
`disable-model-invocation`	`true`	Never picked automatically; it must be selected manually

Resolve conflicting agent definitions

With duplicate names, the lowest-level agent wins. So a repository agent overrides an organization agent, which overrides an enterprise agent. An assigned custom agent uses the latest profile version for its repository and branch. Work in the pull request it opens keeps that same version for consistency.

EXAM TRAP

An enterprise-level agent does not beat a repository agent with the same name. The repository-level definition takes precedence.

Catch contradictory outputs

Review and combine subagent results before you merge. A coordinated workflow ends with that review step. Copilot CLI's built-in rubber-duck agent gives a second opinion. The GitHub Copilot app lets you inspect generated changes, CI results, reviews, pull request status and session history before merging.

Resume and steer workflows

A resumed dynamic workflow can reuse finished work. It reuses saved results from completed steps and subagents instead of starting a new run. Dynamic workflows in Copilot CLI need experimental features, turned on with `--experimental` or `/experimental on`. When you steer a cloud agent session, Copilot applies the input after its current tool call.

Key takeaways

- The orchestrator, usually the main agent, tracks the goal and delegates to subagents.
- Use `/fleet` for independent tasks with clear boundaries, not for sequential ones.
- `/fleet` can use more AI Credits, since each subagent calls the model on its own.
- Isolate concurrent app sessions in separate git worktrees or cloud sandboxes.
- On a name clash, a repository-level agent overrides organization and enterprise agents.

Sources: Delegating and parallelizing work with agents - GitHub Docs (opens in a new tab) · Custom agents - GitHub Docs (opens in a new tab) · Application card: GitHub Copilot Agents - GitHub Docs (opens in a new tab) · Using dynamic workflows - GitHub Docs (opens in a new tab) · Custom agents configuration - GitHub Docs (opens in a new tab) · Managing agent sessions - GitHub Docs (opens in a new tab)

Configure observability for multi-agent behavior by using logs, artifacts, and operational signals

Start here

THINK FIRST

After several agents work on one change, which records let you reconstruct who did what, and why?

Why it matters: Questions ask which logs, artifacts or audit records to set up so multi-agent work can be reviewed and analyzed later.

Observability means every agent leaves records you can review later. Workflow artifacts keep run output, the audit log keeps agent activity, and synced session history answers questions about past sessions. Each record has gaps, so you combine them.

Key terms

Term	Meaning
Workflow artifact	Files a workflow run produces that persist after the job ends and can be shared with another job.

Term	Meaning
Audit log	The enterprise record of agent activity and Copilot plan changes, kept for **180** days.
Check suite	Created for each workflow run; it holds one check run for each job.
Artifact attestation	A provenance record linking an artifact to its workflow, repository, environment, commit SHA and triggering event.

Make every run leave artifacts

Upload run output as artifacts for later review. Artifacts persist after a job finishes, and other jobs in the same workflow can use them. The `upload-artifact` and `download-artifact` actions move them. Uploaded build and test output helps you debug failed tests or crashes and view test coverage.

EXAM TRAP

Caching is not how you keep build logs for review. Artifacts are the tool for files you need to view after the run ends.

Hand off work between jobs

A job that needs another job's artifacts waits for it. A dependent job waits until the earlier job completes successfully, and the `needs` keyword runs jobs in sequence. With no name given, `download-artifact` fetches every artifact of the run, one directory per artifact. Pulling artifacts from another run needs a token and a run identifier.

Prove artifacts are intact

Attestations and digests show an artifact's origin and integrity. An artifact attestation links the artifact to its workflow, repository, environment, commit SHA and triggering event. When a downloaded artifact's digest differs from the upload step's, the run shows a warning in the UI and in the job logs.

EXAM TRAP

Deleting a workflow run removes all of its artifacts from storage. Keep the run until review and audit are done.

Trace agent activity in the audit log

Search the audit log with `actor:Copilot`. The enterprise audit log records agent activity on the GitHub website and changes to the Copilot plan. It keeps events for **180** days. For long-term history and alerts on unusual activity, stream it to a SIEM platform.

EXAM TRAP

The audit log does not hold the prompts a user types to Copilot locally. Capture them with a custom solution, such as hooks feeding your own logging service.

Record session events with hooks

Hooks can log what happens in a local session. For example, a `sessionStart` hook can write each session's start date to a log file. To monitor and steer a Copilot CLI session from elsewhere, enable remote control. The session keeps running on the original machine, so that machine must stay online.

Analyze behavior after the fact

Ask Copilot about your synced session history. Synced history covers Copilot in the cloud, code review, Copilot CLI, VS Code, JetBrains IDEs and the GitHub Copilot app. You can query only sessions you started. Syncing must be enabled and allowed by your organization's policies.

Records for post-hoc analysis and their limits

Record	What it captures	Limit
Workflow artifacts	Files a run produces, kept after jobs end	Deleted with the workflow run
Audit log	Agent activity on the website and Copilot plan changes	Keeps **180** days; no local prompts
Log archive after a partial re-run	Logs of the re-run jobs only	Earlier attempts' archives hold the rest
Synced session history	Past sessions across Copilot surfaces	Only sessions you started

Document outcomes on the pull request

Pull request summaries record what changed. Copilot can write a summary of changes in a pull request description or comment. It helps reviewers, but it is not a code review. To request changes, mention `@copilot` in a comment or push commits to the branch.

EXAM TRAP

Your own approval of a Copilot pull request does not count toward required approvals. Another reviewer must approve it before merging.

Key takeaways

- Keep run output as artifacts for review; deleting the run deletes them.
- Search the audit log with `actor:Copilot`; it keeps events for **180** days.
- Local prompts are not in the audit log, so capture them with hooks.
- By default, workflows do not run automatically when Copilot pushes to a pull request.
- After a partial re-run, download earlier attempts' log archives for a full set.

Sources: Workflow artifacts - GitHub Docs (opens in a new tab) · Reviewing audit logs for GitHub Copilot - GitHub Docs (opens in a new tab) · GitHub Copilot on GitHub.com and GitHub Mobile - GitHub Docs (opens in a new tab) · Using workflow run logs - GitHub Docs (opens in a new tab) · Store and share data with workflow artifacts - GitHub Docs (opens in a new tab) · Customize agent workflows with hooks - GitHub Docs (opens in a new tab) · Review output from Copilot - GitHub Docs (opens in a new tab)

Detect and respond to multi-agent failures and degraded behavior

Start here

THINK FIRST

When one agent stalls, fails halfway or drifts off course, how do you notice it and recover safely?

Why it matters: Questions describe a stuck, failed or misbehaving agent session and ask how to detect it, steer it or recover from it.

Watch sessions through their live logs so you spot a stalled or failing run early. You can steer a drifting session, retry a stuck task by reassigning its issue, or stop a run and keep its pushed commits. A human stays in the loop, because agent pull requests need human review and merge.

Key terms

Term	Meaning
Session log	A live view of an agent's work, with its progress, token usage and session length.
Steering	A message sent while Copilot works, applied to the current task instead of starting a new one.
Queued message	A message that waits for the current task to finish, then runs as the next turn.
Agent Merge	Copilot watching a pull request and fixing the blockers you pick: failed workflows, merge conflicts, or both.
Agent hook	A check you configure to run around an agent's tool calls, such as before or after a tool runs.

Spot a session that never started

An assigned issue should get a reaction, then a draft pull request. After you assign an issue, Copilot leaves an eyes reaction and soon opens a linked draft pull request. A "Copilot started work" event offers View session, whose logs stream live. If the task is stuck, unassign the issue and assign it to Copilot again.

Know the hard limits

A session on GitHub Actions stops after 59 minutes. That limit cannot be extended or bypassed, so a longer task stops partway. Agent hooks time out after **30** seconds by default; raise `timeoutSec` if needed. Copilot cloud agent uses a hooks file only when it sits on the default branch.

EXAM TRAP

An invalid model provider configuration does not fall back to GitHub-hosted models. Copilot CLI exits with an error instead.

Diagnose blocked work

Firewall warnings name the blocked address and command. When the Copilot firewall blocks a request, the warning shows where it went and which command sent it. A ruleset or branch protection rule can also stop Copilot from creating or updating pull requests; administrators can add a bypass. If Copilot stalls on a pull request comment, post that comment again.

Steer a drifting session

Steering corrects a session without restarting it. In a cloud agent session, Copilot acts on your steering input once its current tool call ends. Every steering message uses AI credits, and third-party coding agents cannot be steered.

Steering and queuing in Copilot CLI compared

Feature	Steering	Queued message
How you send it	A plain prompt while Copilot is thinking (the default)	Ctrl+Enter or Ctrl+Q
When it takes effect	Within the task in progress	As the next turn, after the task finishes

Stop and contain a session

Stopping a session keeps the commits already pushed. Stopping ends the GitHub Actions run, and the partial work stays for review. In Copilot CLI, press Esc twice to interrupt the current operation if it starts acting unexpectedly. Copilot cloud agent pushes to one branch only: the existing pull request branch when triggered with `@copilot`, otherwise a new `copilot/` branch.

EXAM TRAP

Do not assume concurrent app tasks must share the default branch. Each GitHub Copilot app session can use its own git worktree or a cloud sandbox.

Trace changes back to sessions

Every agent commit links to its session logs. Each commit message carries that link, for code review and auditing. The logs show the work done and the tools used. They do not replace your own review and testing, so still check the changes yourself.

Keep a human in the loop

A human must review and merge agent pull requests. Copilot cloud agent cannot mark its pull requests Ready for review, approve them or merge them. SDK lifecycle hooks such as `onPreToolUse` and `onPostToolUse` can add safety checks and approval workflows around tool execution.

EXAM TRAP

Workflows do not run on their own on an agent pull request. A user with write access must approve them, which also stops chains of automations.

Recover blocked pull requests

Agent Merge works on failed workflows and merge conflicts. Turn it on with Watch and fix in the merge box, then choose the blockers Copilot should handle. Subagents suit isolated tasks: each works in its own context window and returns a final result. They use the main session's tools and model but cannot create more subagents.

EXAM TRAP

Copilot code review does not catch every problem, especially in large or complex changes. Back it up with careful human review.

Key takeaways

- Watch the live session log to spot stalled or failing work early.
- A session on GitHub Actions stops at **59** minutes and cannot be extended.
- Stopping a session keeps the commits it already pushed.
- Retry a stuck task by unassigning the issue and assigning Copilot again.
- Agent pull requests need a human to review and merge them.

Sources: Troubleshooting GitHub Copilot cloud agent - GitHub Docs (opens in a new tab) · Managing agent sessions - GitHub Docs (opens in a new tab) · Risks and mitigations for GitHub Copilot cloud agent - GitHub Docs (opens in a new tab) · Steering agents in GitHub Copilot CLI - GitHub Docs (opens in a new tab) · GitHub Copilot on GitHub.com and GitHub Mobile - GitHub Docs (opens in a new tab) · Application card: GitHub Copilot Agents - GitHub Docs (opens in a new tab) · Customize agent workflows with hooks - GitHub Docs (opens in a new tab) · Using subagents in your IDE - GitHub Docs (opens in a new tab)

Manage the lifecycle of agents within multi-agent workflows

Start here

THINK FIRST

How do you add, change or retire an agent in a running workflow without breaking it or losing its history?

Why it matters: Questions describe a change to an agent in a workflow and ask which setting, command or step keeps work safe and traceable.

An agentic workflow is a file in the repository, so its agents are managed like code. You add or swap an agent in the frontmatter, revise saved definitions for future runs, and disable rather than delete to retire. Run logs let you review what each agent run did.

Key terms

Term	Meaning
Frontmatter	YAML between --- markers that sets a workflow's triggers, permissions and allowed write operations.
Engine	The frontmatter property that picks the coding agent; GitHub Copilot is the default.
Safe outputs	Validated write operations declared in the frontmatter; the only way an agentic workflow can write.
Dynamic workflow	A Copilot CLI workflow that runs subagents and keeps its definition in an extension.
Extension	Where a dynamic workflow's definition is saved: for a session, in a personal directory or in a repository.

Anatomy of an agentic workflow

An agentic workflow is a markdown file with YAML frontmatter. The frontmatter sets when it runs, its permissions and its allowed writes. The markdown body holds plain-language instructions the agent follows. You compile the file into a `.lock.yml` Actions workflow, then commit and push both files to the default branch.

EXAM TRAP

Wrong: commit only the compiled `.lock.yml` file. The markdown file and its compiled `.lock.yml` file both go to the default branch.

Choosing the coding agent

The `engine` frontmatter property picks the coding agent. Copilot runs when no engine is set. Supported agents include GitHub Copilot, which needs a Copilot plan, plus Anthropic Claude, OpenAI Codex and Google Gemini. To replace the agent, change `engine`. Every engine needs its own authentication secret set up in the repository.

Adding agents for everyone

Organization-level custom agents live in an `/agents` directory. That directory sits in the organization's `.github` or `.github-private` repository. Every member can use these agents, even without access to that repository. In an enterprise, owners can restrict custom agents with a ruleset. To share a dynamic workflow, copy the folder holding its `extension.mjs` into `.github/extensions/`.

EXAM TRAP

Wrong: members need access to the `.github-private` repository to use its agents. Organization-level custom agents reach all members either way.

Guardrails for added agents

New agentic workflows start with read-only repository permissions. Writes such as opening issues, posting comments or creating pull requests go only through validated safe outputs in the frontmatter. Role-based access controls limit who can trigger or modify the workflow. A dynamic workflow's subagents inherit the permission grants of the session that launched them.

EXAM TRAP

Wrong: subagents start with no permissions of their own. A dynamic workflow's subagents inherit whatever the starting session has already allowed.

Handing data between agents

Pass data that later steps or jobs need as explicit outputs. Do not leave it only in logs. Pick the output type by where the consumer runs.

Step outputs and job outputs compared

Feature	Step outputs	Job outputs
Passes values	Between steps in the same job	Across different jobs
Relies on	Steps sharing one job	Job dependencies

Revising a dynamic workflow

Revise the saved definition, not a one-run override. Test a new workflow on a small scope first, such as **two or three** files, then ask Copilot to revise it if needed. Revisions are saved to its extension. New runs use them once the updated extension loads, and a paused run keeps its saved results. Copying a session workflow shares its definition, not its run history.

EXAM TRAP

Wrong: teammates get a revision automatically. For a workflow shared in a project repository, people must pull the updated repository and reload their extensions first.

Retiring a workflow

Disable a workflow to stop triggers while keeping its file. The definition stays in the repository, and you can re-enable it later. Use `gh workflow disable` with the workflow's name, ID or file name. Existing runs are not cancelled. In a public repository, scheduled workflows also disable themselves after **60** days without activity.

EXAM TRAP

Wrong: disabling a workflow cancels its runs. Existing runs are kept, but you cannot re-run them or their jobs until you enable the workflow again.

Auditing agent activity

Review agent runs with the `gh aw` commands. The logs command lists recent runs with duration, token usage and AI Credit (AIC) estimates. The audit command inspects one run, including its estimated inference cost. In enterprise managed settings, the identity capture class ties activity to a user without collecting prompts or responses.

Example: Lists recent agentic workflow runs, then inspects a single run in detail.

```
gh aw logs
gh aw audit RUN-ID
```

- `RUN-ID`: the ID of the run to inspect

Key takeaways

- Add or replace a workflow's coding agent with the `engine` frontmatter property; Copilot is the default.
- Agentic workflows are read-only by default, and writes happen only through declared safe outputs.
- Change future runs by revising the saved definition; a paused run keeps its saved results.
- Retire a workflow by disabling it: the file stays, and you can re-enable it later.
- Use `gh aw logs` to review recent runs and `gh aw audit` to inspect one.

Sources: GitHub Agentic Workflows - GitHub Docs (opens in a new tab) · Dynamic workflows - GitHub Docs (opens in a new tab) · Disabling and enabling a workflow - GitHub Docs (opens in a new tab) · Preparing to use custom agents in your organization - GitHub Docs (opens in a new tab) · Using dynamic workflows - GitHub Docs (opens in a new tab) · Build reliable workflows - outputs, contexts, triggers, and cross-job handoffs - Training | Microsoft Learn (opens in a new tab) · Monitoring agentic activity in your enterprise - GitHub Docs (opens in a new tab)

Domain 6 | 10–15% of the exam | 9 min read

Implement guardrails and accountability

Overview

Every point below comes from GitHub's official documentation; each section lists its sources.

Define autonomy levels

Start here

THINK FIRST

How much should an AI agent be allowed to do on its own, and what decides where humans step in?

Why it matters: Questions describe an agent task and its risks and ask which level of autonomy, permission, or review fits.

Match autonomy to risk. Give each agent only the access its task needs, and take extra care where its actions cannot be undone or carry high stakes. On GitHub, agents propose changes in pull requests that people review before merging.

Key terms

Term	Meaning
Assistant	An AI that gives suggestions or explanations, takes no repository actions, and leaves each step to the user.
Agent	An AI that interprets a goal, picks intermediate steps, uses tools, and acts inside a workflow.
Least privilege	Granting only the access a task requires, not broad access.
Privileged trigger	A workflow trigger, such as pull_request_target, that may get write access and secrets.
Ephemeral runner	A runner that starts clean and is discarded after use, so it cannot stay compromised.

Assistant or agent

An assistant returns control; an agent takes action in a workflow. An assistant responds to a prompt, produces output and hands control back to the user. An agent interprets a goal, decides the intermediate steps and uses tools to act inside a workflow.

Assistant and agent behavior compared

Feature	Assistant	Agent
Output	Suggestions or explanations	Actions taken inside a workflow
Planning	Responds to one prompt	Interprets a goal and decides intermediate steps
Repository actions	None; the user applies each step by hand	Uses tools to make changes

Classify actions by risk

Irreversible or high-stakes actions need the most caution. Ask whether an action can be undone and how much harm it could cause. Apps holding sensitive data or allowing high-impact actions should not get Always allow. Enterprise administrators can turn off computer use in managed settings, and users cannot override that locally.

EXAM TRAP

Always allow is not a safe shortcut for every app. Avoid it where the app holds sensitive information or supports high-impact actions.

Scope access narrowly

Give each session only the tools and permissions its task needs. Avoid broad access when narrow scoping works. For event-triggered automations, pick the exact tools allowed, which limits damage from prompt injection in untrusted input. Copilot CLI starts with file access scoped to its launch directory, and the permission mode decides when it asks first.

Humans review through pull requests

Agents propose changes on a branch and wait for review before merge. By default, workflows on Copilot cloud agent code run only after a write-access user clicks Approve and run workflows. You can also stop Actions workflows from creating or approving pull requests, since unchecked merges are a security risk.

EXAM TRAP

Copilot code review does not replace people. Use it to supplement human reviews.

Limit untrusted triggers

By default, automations ignore events from users without write access. A setting lets you opt in. In the cloud, Copilot is read-only for users with only read access. Use pull_request_target or workflow_run only when a workflow truly needs privileged context; workflow_run separates privileges better.

EXAM TRAP

Running pull_request_target or workflow_run with a checkout of an untrusted pull request is dangerous. Attackers can use it to take over the repository.

Protect secrets and tokens

Anyone with write access can read every repository secret. So keep workflow credentials at least privilege. Set the default GITHUB_TOKEN to read-only repository contents and raise it per job. OIDC lets workflows reach a cloud provider without long-lived secrets. Required reviewers keep environment secrets locked until someone approves.

Example: Sets the default token permission to read-only access for repository contents.

```
permissions:
  contents: read
```

Isolate where agents run

GitHub-hosted runners start clean; self-hosted ones can stay compromised. Self-hosted runners should almost never serve public repositories, because anyone can open a pull request and compromise them. The cloud agent also has a firewall on by default to stop code or data exfiltration.

Runner types compared for isolation

Feature	GitHub-hosted	Self-hosted	Just-in-time (JIT)
Clean virtual machine	Yes, ephemeral and isolated	Not guaranteed	Ephemeral self-hosted runner
Persistence	Cannot be persistently compromised	Untrusted workflow code can compromise it	Removed after at most **1** job

Audit and Responsible AI

GitHub follows six responsible AI principles. They cover transparency and accountability, plus fairness, inclusiveness, reliability and safety, and privacy and security. Audit logs record each action, its time, and the account behind it. SDK hooks like onPreToolUse add safety checks and approvals, and CODEOWNERS on `.github/workflows` forces review of workflow changes.

Key takeaways

- An assistant suggests and returns control; an agent picks steps, uses tools, and acts in a workflow.
- Take extra care where agent actions are irreversible or highly consequential.
- Give each session only the tools, agents, and permissions its task requires.
- Copilot code review adds to human review; it never replaces it.
- Agent pull requests need a write-access user's approval before their workflows run.

Sources: Application card: GitHub Copilot Agents - GitHub Docs (opens in a new tab) · Define agentic AI in the SDLC - Training | Microsoft Learn (opens in a new tab) · Secure use reference - GitHub Docs (opens in a new tab) · Risks and mitigations for GitHub Copilot cloud agent - GitHub Docs (opens in a new tab) · Secure use reference - GitHub Docs (opens in a new tab)

Implement guardrails and human-in-the-loop workflows

Start here

THINK FIRST

Which automated actions deserve a human approval, and which should simply run with tight permissions?

Why it matters: Questions describe a risky automation and ask which approval, policy, or permission setting contains it without slowing work.

Guardrails split actions into those that need a person and those that can run alone. Risky paths, such as merging pull requests or reaching secrets, get reviewers and policy. Everything else runs with the smallest token permissions, scoped by risk so approvals stay few.

Key terms

Term	Meaning
Workflow execution protections	Policies that decide which workflow runs may execute; when enforced, disallowed runs fail.
Actor rules	Rules on who can trigger workflows: users, repository roles, GitHub Apps, Copilot, and Dependabot.
Evaluate status	A policy enforcement status that shows when users would hit a restriction, without blocking them.
GITHUB_TOKEN	A GitHub App installation access token used by a workflow, limited to its own repository.
OIDC token	A token a job fetches with `id-token: write`; the cloud provider checks its claims before trusting it.

Actions that need human judgment

Reserve human review for merges, secret access, and workflow edits. Required reviewers can gate environment secrets, so a job waits for approval before reading them. Listing `.github/workflows` in CODEOWNERS means any change to workflow files needs a designated reviewer first. You can also stop GitHub Actions from creating or approving pull requests.

EXAM TRAP

Letting workflows create and approve their own pull requests is not a safe shortcut. A pull request merged without proper oversight is a security risk.

A controlled path for bot pull requests

Runs from pull requests made with `GITHUB_TOKEN` wait for approval. When such a workflow opens or updates a pull request, the new runs start in an approval-required state. A user with write access starts them with Approve workflows to run. Using a GitHub App installation token or a personal access token instead skips that wait.

Blocking disallowed runs

Enforced execution protections fail disallowed runs with an error. Central policy overrides any single misconfigured workflow file. Actor rules then decide who may trigger workflows, covering individual users, repository roles, GitHub Apps, Copilot, and Dependabot.

> **EXAM TRAP**
> Write access does not have to include running CI. It does by default, but actor rules can grant write access without the right to execute workflows.

Keeping approvals proportional

Match protection to risk, then test policies in Evaluate mode. Production deploy workflows may need more protection than minor automations. So scope policies to specific workflow paths or required workflows. Policy insights then show whether a policy works as intended or adds needless friction.

Active and Evaluate policies compared

Feature	Active (enforced)	Evaluate
Disallowed runs	Fail with an error	Are not blocked
Policy insights show	Runs that were blocked	Runs that would have been blocked

Least-privilege GITHUB_TOKEN

Limit the token's permissions, because any action can reach it. The token only covers the repository that holds the workflow. Permissions start from the enterprise, organization, or repository default. The workflow level, then the job level, adjusts them. Organization owners can also restrict its write access per repository.

> **EXAM TRAP**
> Not passing the token to an action does not keep it away. Actions can read it through the `github.token` context, so limit its permissions.

Writing the permissions key

Naming one permission sets every unnamed permission to none. Here `write` also grants `read`. Use `permissions: {}` to turn off every permission. A job needs `id-token: write` to fetch an OIDC token. The `copilot-requests` permission accepts only `write` or `none`.

Example: Grants only the right to fetch an OIDC token; every other permission becomes none.

```
permissions:
  id-token: write
```

- `id-token: write`: required to fetch an OpenID Connect token

Token access by trigger event

The trigger event can raise or lower default token access. Check what started a workflow before trusting its default permissions. Fork and Dependabot runs are cut back, but one trigger keeps write access.

Default GITHUB_TOKEN access by trigger

Trigger	Default token access	Exception or extra
`pull_request_target`	Read/write repository permission	Applies even when triggered from a public fork
Other pull request event from a fork	Write permissions reduced to read-only	Unless Send write tokens to workflows from pull requests is selected
Dependabot pull request	Read-only	Only Dependabot secrets; the permissions key can raise access

EXAM TRAP

A `pull_request_target` run from a public fork is not read-only. Its token still gets read/write repository permission.

Safer credentials and inputs

OIDC tokens are short-lived and checked against a trust definition. With OIDC, the cloud provider issues a token valid for one job, which then expires. It first matches the token's subject and claims to the cloud role's trust definition. Anyone with write access can read every repository secret, so stored credentials need minimal privileges.

EXAM TRAP

Values like `github.event.issue.title` are not trusted input. Attackers can place malicious content there, so handle them carefully to avoid script injection.

Key takeaways

- Enforced workflow execution protections make disallowed runs fail with an error.
- Try a new policy in Evaluate status and check policy insights before enforcing it.
- Once the `permissions` key names one permission, every unnamed permission becomes none.
- Runs triggered by `pull_request_target` get a read/write token, even from a public fork.
- Required reviewers keep a job from reading environment secrets until someone approves.

Sources: Secure use reference - GitHub Docs (opens in a new tab) · Use GITHUB_TOKEN for authentication in workflows - GitHub Docs (opens in a new tab) · Governing execution of GitHub Actions workflows - GitHub

Docs (opens in a new tab) · About Actions policies - GitHub Docs (opens in a new tab) · GITHUB_TOKEN - GitHub Docs (opens in a new tab) · Workflow syntax for GitHub Actions - GitHub Docs (opens in a new tab) · OpenID Connect - GitHub Docs (opens in a new tab) · Script injections - GitHub Docs (opens in a new tab)

MockCert

Take a GitHub Certified: Agentic AI Developer mock exam the way the real one feels

5 timed mock exams · 60 questions in 120 minutes each · one is free

- Free to start: study notes, explained quizzes, flashcards and the first mock exam
- Timed mock exams built to the official exam guide
- A verifiable Certificate of Assessment when you pass a premium mock

Explore GitHub Certified: Agentic AI Developer on MockCert



Open it on your phone

mockcert.com/github-gh-600/

Scan the code with your phone's camera.