SPARQIO

GH-600 MASTERCLASS · STUDY NOTES

MODULE 0

Orientation & the GH-600 Landscape

Design, operate, and govern AI coding agents in the GitHub SDLC

Module 0 — Orientation & the GH-600 Landscape

GH-600 Agentic AI Developer Masterclass · Study Guide **Domain focus:** Overview / exam mechanics
Est. learner time: 45–60 minutes

> Independent study material. Not affiliated with, endorsed by, or sponsored by GitHub or Microsoft. Contains no exam questions or dumps. Every claim is grounded in official Microsoft Learn and GitHub documentation — see **Sources**.

Learning objectives

By the end of this module you can:

- State what the GH-600 credential actually certifies and who it is for.
- Recite the current exam facts and, more importantly, know **where to verify them** before you book.
- Name the six skills-measured domains and their approximate weights.
- Explain the one mental model the whole exam is built on: `plan → act → evaluate`.
- Explain why GitHub is treated as the **system of record and control plane**.
- Choose a study path (4-week or 10-day) and stand up your practice sandbox.
- Study the right way: official docs first, no exam dumps, ever.

Prerequisites

- A GitHub account and the `git` and `gh` CLIs installed.
- Working familiarity with pull requests, branches, and GitHub Actions at a basic level. You do not need to be an expert — the course builds that.

---

1. What GH-600 actually certifies

GH-600 leads to the credential **"GitHub Certified: Agentic AI Developer."** It is an **intermediate** certification on the GitHub product, and it is not a prompt-engineering trivia test.

The exam validates that you can **design, operate, evaluate, and govern AI coding agents inside GitHub-centered software-development workflows** (S1, S2). Read that sentence twice. The verbs — design, operate, evaluate, govern — are the job. You are being certified as an **operator of production**

agent systems, not as someone who can coax a clever answer out of a chatbot.

The role profiles Microsoft lists for this credential make the point: AI Engineer, App Maker, Data Engineer, Developer, DevOps Engineer, and Solution Architect (S1). What those roles share is responsibility for systems that ship. The exam rewards the instincts of someone who has to keep a real pipeline safe.

The operator's mindset

Throughout the course, when a scenario is ambiguous, ask what a careful operator of a production system would do. Concretely, the exam expects you to:

- define agent tasks clearly, with explicit inputs, outputs, and success criteria;
- separate planning from execution;
- scope tools and permissions to least privilege;
- configure MCP and tool access deliberately;
- preserve inspectable artifacts (plans, PRs, logs, traces);
- manage memory and state so context does not drift;
- evaluate failures from evidence, not vibes;
- coordinate multiple agents without collisions;
- apply guardrails and human-in-the-loop controls **without killing delivery speed** (S2).

That last clause matters. This exam is not "add approvals everywhere." It is "add the right controls in the right places."

---

2. Exam mechanics (and how to verify them)

Here are the exam facts as documented at the time this course was written. Treat them as a **starting point to verify**, not gospel — certification details change, and this exam has a beta/GA transition.

Item	Detail
Exam code	GH-600
Credential	GitHub Certified: Agentic AI Developer
Focus	Developing in Agentic AI Systems
Duration	120 minutes
Passing score	700 (on Microsoft's 1–1000 scale)
Level	Intermediate
Product	GitHub
Platform note	Microsoft provides the exam platform; GitHub owns the exam/certification content

Source: Microsoft Learn certification page (S1) and official study guide (S2).

The beta/GA caveat — read this before booking

At the time of writing, the certification page still labels GH-600 as **beta**. Community notes indicate the beta window ran through **May 31, 2026**, with general availability from **July 2026**. Do not trust that date range from any course, blog, or study repo — including this one.

Rule: before you book, open the official Microsoft Learn certification page (S1) and the Pearson VUE registration page and confirm the current status, duration, price, and passing score yourself. Certification logistics are exactly the kind of fact that goes stale. The professional move is to verify at the source.

---

3. The six domains and their weights

The exam is organized into six skills-measured domains. Study time should track exam weight — that is why this course allocates the most depth to Domain 2.

#	Domain	Weight	Course module
1	Prepare agent architecture and SDLC processes	15–20%	Module 1
2	Implement tool use and environment interaction	20–25%	Module 2
3	Manage memory, state, and execution	10–15%	Module 3
4	Perform evaluation, error analysis, and tuning	15–20%	Module 4
5	Orchestrate multi-agent coordination	15–20%	Module 5
6	Implement guardrails and accountability	10–15%	Module 6

Source: official GH-600 study guide (S2).

A recommended study-time split, weighted for difficulty as well as exam weight: Domain 2 (~25%), Domain 1 (~20%), Domain 4 (~18%), Domain 5 (~17%), Domain 6 (~12%), Domain 3 (~8%). Domain 2 is both the heaviest and the most hands-on, so it earns the most lab time.

---

4. The one mental model: plan → act → evaluate

If you internalize one thing from this module, make it this loop:

```
plan → act → evaluate
```

An agent should produce an **inspectable plan before it acts**, act only inside **approved boundaries**, and produce outputs that can be **reviewed and evaluated** (S4, S5). Almost every correct exam answer is a specific application of this loop:

- *Plan* — force a structured plan artifact before any write action; validate and approve it. (Domain 1)
- *Act* — give the agent the smallest sufficient tool set, scoped to a repo and a branch, with safe execution paths. (Domains 2, 3)
- *Evaluate* — judge output against success criteria from logs, traces, and artifacts; classify root causes; tune from evidence. (Domain 4)

Multi-agent coordination (Domain 5) and guardrails (Domain 6) are what you add when the loop runs at scale and with real risk.

When you meet a hard question, locate it on the loop. "The agent acted without explaining intent" is a *plan* failure. "The agent used a destructive command" is an *act*/tool failure. "The agent used a stale issue comment as context" is an *evaluate*/context failure. The loop is your triage tool.

---

5. GitHub as the system of record and control plane

The second unlocking idea: on this exam, **GitHub is where agent work is governed, recorded, and audited.** Plans live in PR bodies and issue comments. Actions happen on branches. Approvals happen through branch protection, CODEOWNERS, and environment gates. Evidence lives in workflow logs, uploaded artifacts, and check-run summaries (S2, S7, S11).

So when an answer choice hides work outside GitHub — "trust the agent," "keep the plan in chat," "grant broad access and skip review" — it is almost always wrong. The high-yield answers keep GitHub as the **control plane**: scoped permissions, branch-based work, PR review, required checks, inspectable artifacts, environment approval for sensitive actions.

Your stack analogy

If you already operate an agentic workflow of your own, map it: your orchestrator is the coordinator, your source-of-truth notes are durable memory, your PRs and logs are inspectable artifacts, your approval gates are guardrails. GH-600 rewards you for **formalizing that instinct in GitHub terms**.

---

6. How to study (the honest version)

There are two supported paths in this course. Pick by how much time you have.

4-week plan — one domain cluster per week:

1. Week 1 — foundations & vocabulary (this module + Domain 1).

2. Week 2 — tool use, MCP, execution environments (Domain 2, the heavy one).
3. Week 3 — architecture, memory, guardrails (Domains 1, 3, 6).
4. Week 4 — evaluation, multi-agent, and a timed mock (Domains 4, 5).

10-day sprint — compressed: certification page → foundations → architecture → tooling/MCP → sandbox labs → memory/state → evaluation → multi-agent → guardrails → timed mock + gap review.

Either way, the method is the same four moves:

1. **Study the six official domains** from the study guide (S2).
2. **Build a sandbox repo** and practice each domain with real GitHub primitives.
3. **Do the MCP/tooling labs** — Domain 2 is the heaviest, so over-invest there.
4. **Drill flashcards** until the vocabulary is automatic, then take a **timed mock** and review every miss against the official source.

The one rule that protects your certification

No exam dumps. Ever. Do not study from "brain dumps," leaked questions, or sites promising real exam items. It violates the certification agreement, it teaches you nothing durable, and it puts your credential at risk. This course contains zero exam questions — every practice item is original and written to a documented skill. When you miss a question, you review the **official doc**, not a dump. That is also simply how you actually learn this material.

---

7. Set up your sandbox (Lab 0)

You cannot learn agent governance by reading. You need a repo you can break. Module 0's lab creates a private `gh600-sandbox` repository — your practice ground for every later module. Do the lab now; the rest of the course assumes it exists. Full steps are in `50_lab.md`.

Recap

- GH-600 certifies you as an **operator** of AI coding agents in the GitHub SDLC — design, operate, evaluate, govern.
- Verify exam facts (beta/GA, duration, score) at the **official source** before booking.
- Six weighted domains; Domain 2 (tools/MCP) is heaviest — invest there.
- The whole exam is `plan → act → evaluate`, with **GitHub as the control plane**.
- Study from official docs, build a sandbox, drill vocabulary, take a timed mock.
- **No dumps.** Original practice + official review only.

Sources

- **S1** — GitHub Certified: Agentic AI Developer (certification page) — https://learn.microsoft.com/en-us/credentials/certifications/agentic-ai-developer/
- **S2** — GH-600 study guide (skills measured, domains, weights) — https://learn.microsoft.com/en-us/credentials/certifications/resources/study-guides/gh-600
- **S4** — Foundations of Agentic AI (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/foundations-agentic-ai/
- **S5** — Design agent architecture & SDLC integration (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/design-agent-architecture-integration/
- **S7** — Prepare for custom agents (GitHub Docs) — https://docs.github.com/en/copilot/how-tos/administer-copilot/manage-for-organization/prepare-for-custom-agents
- **S11** — Build guardrails (GitHub Docs) — https://docs.github.com/en/copilot/tutorials/cloud-agent/build-guardrails

Where this appears on the exam

Module 0 is orientation, not a scored domain — but the two ideas here (`plan → act → evaluate` and GitHub-as-control-plane) are the lens for every scored question. If a choice hides intent, skips review, or grants broad unscoped access, it is almost certainly wrong.

SPARQIO

GH-600 MASTERCLASS · STUDY NOTES

MODULE 1

Agent Architecture & SDLC Integration

Design, operate, and govern AI coding agents in the GitHub SDLC

Module 1 — Agent Architecture & SDLC Integration

GH-600 Agentic AI Developer Masterclass · Study Guide **Domain focus:** Domain 1 — Prepare agent architecture and SDLC processes (15–20%) **Est. learner time:** 2.5–3 hours (guide + video + Lab 1 + quiz)

> Independent study material. Not affiliated with, endorsed by, or sponsored by GitHub or Microsoft. Contains no exam questions or dumps. Every claim is grounded in official Microsoft Learn and GitHub documentation — see **Sources**.

Learning objectives

By the end of this module you can:

- Write an **agent task contract** — explicit inputs, outputs, and success criteria — and tell a bad task from a better one (S2, S4).
- Explain why **planning is separated from execution**, and why inspectability is the spine of Domain 1 (S5).
- Produce a **structured plan artifact** in the right GitHub surface: PR body, issue comment, workflow artifact, JSON plan, or check-run summary (S5, S10).
- **Validate and gate** a plan so no write action fires until the plan is checked and approved (S5, S10).
- Recognize the **seven agent anti-patterns** and apply the documented fix for each (S2).
- Assign a **degree of autonomy** and insert **human-in-the-loop** controls without slowing delivery (S2, S7).
- Configure **observability and inspectable artifacts** inside standard dev tooling (S2, S5).

Prerequisites

- Module 0 complete, including the private `gh600-sandbox` repo from Lab 0.
- `git` and the GitHub CLI (`gh`), plus basic fluency with pull requests, branch protection, and GitHub Actions.

---

1. The agent task contract

Domain 1 opens with a discipline you already apply to human tickets: **do not start work you cannot define.** Every agent task needs a contract with three parts (S2, S4).

Contract item	Meaning	Examples
Input	What the agent receives	issue, spec, failing test, roadmap item, error log
Output	What the agent must produce	plan, branch, PR, artifact, report, test result
Success criteria	How acceptance is judged	tests pass, no secret leaks, PR reviewed, lint clean, no public API change

The contract is not paperwork. It is what makes the run **evaluable** later — you cannot judge an output against criteria you never wrote down. This is the same `plan → act → evaluate` loop from Module 0: the contract defines the *plan* and the *evaluate* ends before the agent *acts* (S5).

Bad task vs better task

A bad task is a wish:

```
Improve auth.
```

There is no input scope, no defined output, and nothing to accept against. The agent will guess intent, touch whatever it likes, and produce a diff nobody can grade. A better task is a contract:

```
Review the auth module. Produce a plan first. Then create a feature branch and
a PR that adds session rotation on login. Success criteria: unit tests added,
existing tests pass, no public API change, PR body includes rollback notes.
```

Notice what changed: bounded input (the auth module), an explicit plan-before-act sequence, a defined output (branch + PR), and testable success criteria. On the exam, the "improve auth" style choice is almost always the wrong answer — it is the distractor that hides intent and scope (S2).

---

2. Separate planning from execution — inspectability is the spine

The single highest-yield idea in Domain 1: an agent must **produce an inspectable plan before it acts, act only inside approved boundaries, and produce outputs that can be reviewed** (S5). Planning, reasoning, and action are distinct phases with a boundary between them, and you *configure planning separately from execution* (S5).

Why draw that boundary at all? Because an inspectable plan is the only thing a reviewer — or a gate, or an audit — can act on **before** a write happens. Once the agent has edited files, opened a PR, or run a command, the cheap moment to catch a wrong intent is gone. Inspectability is the spine of this domain: it is what turns an opaque autonomous run into something a careful operator can govern (S5, S2).

State it as a rule: **no write action until the plan has been inspected.** Every control later in this module — plan artifacts, gates, autonomy levels, human-in- the-loop — exists to preserve that rule without grinding delivery to a halt.

---

3. Structured plan artifacts

"Inspectable" is not a vibe; it is an artifact in a place a reviewer already looks. Configure the agent to **output a structured plan** in one of these GitHub-native surfaces (S5, S10):

Artifact	Where it lives	Best for
PR body	the pull request description	change-level plans reviewed alongside the diff
Issue comment	on the driving issue	plans posted before any branch exists
Workflow artifact	uploaded from a GitHub Actions job	machine-produced plans, retained for audit
JSON plan	a structured file (e.g. `plan.json`)	plans a downstream job validates programmatically
Markdown checklist	PR body or artifact	human-scannable step lists with checkboxes
Check-run summary	a check attached to the commit/PR	plan surfaced as a required status a reviewer sees

GitHub's implementation planner pattern makes this concrete: the agent generates a structured, step-by-step plan you can read and approve before implementation begins (S10). The exam rewards you for putting the plan **where governance already happens** — the PR, the check, the artifact — not in transient chat.

A minimal JSON plan an execution job can validate:

```
{
  "task": "add session rotation on login",
  "inputs": ["auth module", "issue #42"],
  "proposed_changes": ["rotate session token in login handler"],
  "files_touched": ["src/auth/session.ts"],
  "tests": ["add rotation unit test", "existing auth tests pass"],
  "risks": ["session invalidation on deploy"],
  "rollback": "revert PR; sessions fall back to prior handler"
}
```

---

4. Validate and gate the plan before write actions

A plan artifact you never check is theater. Domain 1 asks you to **validate agent plans** and **prevent agent action until the plan is checked and approved** (S5, S10). Validation and gating are two moves:

- **Validate** — check the plan against the contract. Does it name inputs, files likely touched, tests, risks, and a rollback? Does its scope match the task, or has it quietly expanded? A JSON plan can be validated by a script; a markdown plan by a reviewer against a checklist (S10).
- **Gate** — insert a control that blocks execution until validation passes. In GitHub terms the gate is concrete: a **planning job** that produces the artifact, an **environment approval or manual gate**, then a separate **execution/evaluation job** that only runs after approval (S5). Branch protection, required checks, and CODEOWNERS review are the same idea applied to the PR itself.

The shape to remember: **plan job → approval gate → execution job.** Planning and execution are different jobs precisely so a human or a check can stand between them. You build exactly this in Lab 1.

---

5. The seven anti-patterns and their fixes

Domain 1 explicitly measures your ability to **identify and mitigate common agent anti-patterns** (S2). Memorize the pattern *and* its fix — the exam tests the fix, not just the diagnosis.

#	Anti-pattern	Why it is bad	Fix
1	Agent acts before planning	Reviewers cannot inspect intent	Require a plan artifact before any write action
2	Blanket write permissions	Increases blast radius	Least privilege + branch/repo scope
3	No durable artifacts	No audit trail	Preserve PRs, logs, traces, uploaded artifacts
4	Tool access not scoped	Agent can touch too much	MCP allow lists, tool permissions, environment gates
5	Memory never pruned	Stale / drifting context	Expiration, pruning, and reset rules
6	No rollback	Failures become expensive to unwind	Defined rollback path + escalation policy
7	Human approves everything	Slows delivery without reducing material risk	Risk-based approvals

Two of these are counter-intuitive and therefore heavily tested. **#5** is a Domain 3 (memory) concern surfacing in architecture — durable memory drifts if it is never pruned. **#7** is the trap that catches over-cautious candidates: adding approval to *every* action, including typo fixes, is an anti-pattern, not a safety

win. The exam is "the right controls in the right places," not "approvals everywhere" (S2, S7).

---

6. Degree of autonomy and human-in-the-loop without slowing delivery

You must **plan and implement a degree of autonomy with guardrails** and **configure human intervention without slowing delivery** (S2). These two requirements are in tension, and resolving the tension is the whole skill.

The move is **risk-based autonomy**: match the amount of human involvement to the blast radius of the action, not to a blanket policy (S2, S7).

- **Low-risk, reversible actions** — drafting a plan, opening a branch, opening a PR for review — can run autonomously. The PR review itself is the safety net.
- **High-risk or irreversible actions** — deploys, touching secrets, changing protected branches, altering security-sensitive code — require explicit human authorization via an environment approval or CODEOWNERS review (S7).

Human-in-the-loop is therefore a **placement decision**, not a volume decision. You place a human where a mistake would be expensive or irreversible, and you let the agent run everywhere else. That is how you keep delivery fast *and* safe. CODEOWNERS scopes review to the files that actually matter; environment gates scope approval to the actions that actually matter (S7). Everything outside those scopes flows without a human touching it.

If an answer choice requires a human to approve a lint fix, it is wrong for the same reason a choice granting broad unscoped write is wrong: both ignore risk.

---

7. Observability and inspectable artifacts

The last Domain 1 skill: **configure observability and control**, and **configure agents to produce inspectable artifacts inside standard development tooling** (S2). "Standard tooling" is the key phrase — you do not build a bespoke dashboard; you make the agent record its work where GitHub already shows it.

Inspectable artifacts include (S2, S5):

- **pull requests** — the diff, plus the plan in the body;
- **issue comments** — intent and decisions, timestamped;
- **workflow logs** — what each job did, retained by Actions;
- **uploaded artifacts** — the plan, test output, reports;
- **check-run summaries** — pass/fail surfaced on the PR;
- **code scanning / secret scanning results** — evidence of safety checks.

The test is simple: if the agent's run happened and left nothing a reviewer can open in GitHub, observability failed. Every action should be traceable to an artifact. This is also what makes Domain 4 evaluation possible later — you can only analyze a failure you have evidence for.

Worked walkthrough — the sandbox governance kit

Lab 1 wires all seven skills into four files in `gh600-sandbox`:

1. `.github/copilot-instructions.md` — the durable contract: plan before acting, branch for changes, PR for review, tests before merge, boundaries on secrets and deploys. This encodes the task-contract discipline as repo memory.
2. `.github/pull_request_template.md` — forces every PR body to carry a structured plan: Plan / Changes / Tests / Risks / Rollback / Human-review. The plan artifact becomes non-optional.
3. `.github/CODEOWNERS` — scopes required human review to sensitive paths, so human-in-the-loop lands exactly where risk lives and nowhere else.
4. `.github/workflows/agent-plan-gate.yml` — a **plan job** (read-only permissions) that uploads a plan artifact, an **approval gate**, then an **execution/evaluation job**. This is planning-separated-from-execution as running code (adapted from Lab F in the study book, grounded in S5/S10).

That kit is a complete, inspectable, risk-gated agent SDLC in one repo — the exact thing Domain 1 asks you to design.

Recap

- Every agent task needs a **contract**: inputs, outputs, success criteria. A bad task is a wish ("improve auth"); a better task is bounded and testable.
- **Separate planning from execution.** Inspectability is the spine — no write action until the plan is inspected.
- Put the **structured plan** in a GitHub surface: PR body, issue comment, workflow artifact, JSON plan, or check-run summary.
- **Validate and gate**: plan job → approval → execution job.
- Know the **seven anti-patterns and their fixes**; #7 (approve everything) is a trap, not a safeguard.
- **Risk-based autonomy**: place humans where blast radius is high, let the agent run everywhere else.
- Produce **inspectable artifacts in standard tooling** so every run is traceable.

Sources

- **S2** — GH-600 study guide (skills measured, Domain 1 bullets, weights) — https://learn.microsoft.com/en-us/credentials/certifications/resources/study-guides/gh-600

- **S4** — Foundations of Agentic AI (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/foundations-agentic-ai/
- **S5** — Design agent architecture & SDLC integration (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/design-agent-architecture-integration/
- **S7** — Prepare for custom agents (GitHub Docs) — https://docs.github.com/en/copilot/how-tos/administer-copilot/manage-for-organization/prepare-for-custom-agents
- **S10** — Implementation planner (GitHub Docs) — https://docs.github.com/en/copilot/tutorials/customization-library/custom-agents/implementation-planner

Where this appears on the exam

Domain 1 is 15–20% of GH-600 — the highest-leverage foundational domain. Expect "safest / best next action" scenarios. Heuristics:

- If a choice **hides intent** (acts with no plan, keeps the plan in chat), it is a *planning* failure — wrong.
- If a choice **grants broad or unscoped write**, it violates least privilege — wrong.
- If a choice **adds approval to a low-risk, reversible action**, it slows delivery without reducing material risk — wrong.
- The right answer usually combines **scoped permissions + branch-based work + PR review + required checks + environment approval for sensitive actions + inspectable plan artifacts** (S2). When in doubt, pick the option that keeps GitHub as the control plane and the plan inspectable before action.

SPARQIO

GH-600 MASTERCLASS · STUDY NOTES

MODULE 2

Tool Use, MCP & Execution Environments

Design, operate, and govern AI coding agents in the GitHub SDLC

Module 2 — Tool Use, MCP & Execution Environments

GH-600 Agentic AI Developer Masterclass · Study Guide **Domain focus:** Domain 2 — Implement tool use and environment interaction (20–25%) — **the heaviest domain on the exam Est. learner time:** 3.5–4 hours (guide + video + Lab 2 + quiz)

> Independent study material. Not affiliated with, endorsed by, or sponsored by GitHub or Microsoft. Contains no exam questions or dumps. Every claim is grounded in official Microsoft Learn and GitHub documentation — see **Sources**.

Learning objectives

By the end of this module you can:

- Choose the **smallest sufficient tool set** for a task and defend the choice with the task → tool table (S2, S6).
- Configure **agent tools** and **tool permissions** at least privilege, so a tool grant never exceeds what the task contract requires (S2, S6, S7).
- Explain what an **MCP server** is, why MCP is the highest-risk grant you can make, and configure servers, **remote MCP**, **MCP registries**, and **MCP allow lists** (S6, S7, S12).
- Run the **execution context checklist** — repo scope, branch scope, PR ability, CI triggering, deploy ability, secrets access, audit evidence — against any agent before it runs (S2, S6).
- Configure an agent to be **invoked in CI** with a scoped `permissions:` block and artifact evidence (S2, S6).
- Apply **branch-based scope** and enable **autonomous branch and PR creation** without ever exposing a protected branch (S2, S6, S11).
- Set **environment-specific constraints** so dev, CI, and production each get their own boundary (S2, S6, S11).
- Design **safe execution paths**: retries, rollback, escalation, partial- failure reporting, and traceability for every action (S2, S6, S12).

Prerequisites

- Modules 0–1 complete: the private `gh600-sandbox` repo exists and carries the Lab 1 governance kit (`copilot-instructions.md` , PR template, CODEOWNERS, plan-gate workflow).
- `git` and the GitHub CLI (`gh`), plus working fluency with GitHub Actions `permissions:` blocks and branch protection.

1. Tool selection — the smallest sufficient tool set

Module 1 taught you to contract the task; this domain decides **what the agent is allowed to touch while executing it**. The governing rule, and the single highest-yield heuristic in the whole domain: the right answer is usually the **smallest sufficient tool set** (S2, S6) — not the most capable configuration, the smallest one that still completes the contract.

Learn this table until you can reproduce it cold (S2, S6):

Task	Tool choice
Read code and propose changes	repo read + branch + PR
Modify code	branch-scoped write, tests, PR
Fetch external context	approved MCP/tool with allow list
Deploy	environment-gated workflow
Access secrets	never direct; GitHub Actions secrets/environments with controls

Two rows carry most of the exam weight. **Fetch external context** is the MCP row: external reach goes through an *approved* server behind an *allow list*, never through an open-ended grant. **Access secrets** is absolute: an agent never holds a secret directly; secrets live in GitHub Actions secrets and environments, released only to gated workflow runs (S6, S11).

The anti-pattern is additive thinking: "give it everything so it won't get blocked." Every unnecessary tool is attack surface, blast radius, and audit noise. The "just in case" grant is Domain 2's blanket write (S2, S12).

2. Configuring tools and tool permissions

Selecting the tool set is half the skill; **configuring** it is the other half. Domain 2 measures both explicitly: *configure agent tools* and *configure agent tool permissions* are separate skills-measured bullets (S2).

The mechanism is layered. Least privilege is applied at every layer, and the effective grant is the **intersection** of all of them (S6, S7):

Layer	Control	Example
Organization	which agents and tools are permitted at all	org readiness policy for custom agents (S7)
Repository	which repos the agent may operate in	repo-scoped agent enablement (S7)
Agent definition	which tools the agent is given	tool list in the custom agent configuration (S8)
Tool	what each tool may do	per-tool permissions, MCP allow list entries (S6)
Run	what a single execution can touch	workflow `permissions:` block, environment gates (S6, S11)

Two consequences follow. First, **a tool grant is a permission decision, not a convenience decision** — treat adding a tool with the same seriousness as granting a human write access (S7). Second, tool permissions are configured *per tool*, not per agent: an agent may hold a read-only search tool and a branch-scoped write tool at the same time, each bounded separately (S6, S8).

The exam phrase to anchor on: **identify required tools first, then configure exactly those, then bound each one** (S2). Selection → configuration → permission — in that order.

---

3. MCP deep dive

3.1 What MCP adds — and why it is high-risk

The Model Context Protocol (MCP) is the standard mechanism for extending an agent with external tools and data: an **MCP server** exposes capabilities — issue search, database queries, external APIs — that the agent can invoke as tools (S6). That is exactly why MCP is the highest-risk grant in this domain: **an MCP server expands the agent's action surface beyond chat and code generation into external systems** (S6, S12). Before MCP, the worst case is a bad diff on a branch. After MCP, the action surface includes whatever the server can reach — and a malicious or compromised server can feed the agent poisoned context or exfiltrate what the agent shows it (S12).

So every MCP decision is a security decision, and the documented controls form a fixed catalogue (S6, S7, S12). Know all seven:

1. **Server registration** — only explicitly registered servers exist for the agent; nothing is discovered ad hoc.
2. **Tool allow list** — of the tools a server exposes, only the listed ones are callable.
3. **Registry controls** — the organization curates *which servers may be registered at all*.
4. **Remote MCP server configuration** — hosted servers are configured deliberately, with identity and transport pinned.
5. **Repository/organization scope** — the server is available where it is needed, not everywhere.

6. **Permissions per tool** — each exposed tool is bounded individually.
7. **Auditability of tool calls** — every invocation leaves evidence you can review (S6, S12).

3.2 Adding an MCP server as a tool to an agent

Adding a server is a skills-measured bullet of its own (S2). The shape of the configuration is consistent: you declare the server (name, type, endpoint or launch command), then declare **which of its tools the agent may call** (S6, S8). A registered server with no allow list is a half-finished configuration — registration answers "does this server exist for the agent," the allow list answers "what, exactly, may be invoked." You do both in Lab 2, and you keep the configuration file in the repo so the grant itself is an inspectable, reviewable artifact (S6, S11).

3.3 Remote MCP configuration

A **remote MCP server** is hosted infrastructure the agent reaches over the network rather than a process on the developer's machine — GitHub's own remote MCP server is the canonical example, giving agents GitHub context (issues, repos, PRs) as tools (S6). Remote changes the risk profile: you are trusting a network endpoint, so configuration must pin *which* server, *what transport*, and *what the server may see*. The exam expects you to know that configuring a GitHub remote MCP server is a distinct, deliberate act (S2, S6) — a remote endpoint is never implicitly trusted just because it is reachable.

3.4 MCP registries

A **MCP registry** is the curation layer: the organization controls which servers are available to be registered in the first place (S6, S7). Registries move the trust decision from every individual developer ("is this server safe?") to a governed org-level review — the same move CODEOWNERS makes for code review. Configure MCP registries is its own skills-measured bullet (S2); the exam signal is that server *supply* is governed upstream, not policed downstream after an incident.

3.5 MCP allow lists

The **MCP allow list** is the finest-grained control and the one the exam leans on hardest: given a registered, trusted server, the allow list names the subset of its tools the agent may actually call (S6). A server may expose twenty tools; if the task contract needs two, the allow list holds two. This is smallest-sufficient-tool-set applied *inside* a single server. When a scenario offers "register the server" versus "register the server and configure an allow list," the second is the answer (S2, S6).

---

4. The execution context checklist

Domain 2's next skill: **evaluate the execution context for an agent** (S2). Execution context means everything about *where and with what authority* the agent runs. The documented checklist is seven questions — run them against every agent, every environment, before it executes (S6):

#	Question	What a good answer looks like
1	Which **repo** can the agent touch?	Named repos only — never org-wide by default
2	Which **branch** can it write to?	A feature branch; protected branches excluded
3	Can it open **PRs**?	Yes — the PR is the review surface, so usually granted
4	Can it trigger **CI**?	Only the intended workflows, with scoped permissions
5	Can it **deploy**?	Only through an environment-gated workflow with approval
6	Can it access **secrets**?	Never directly; via Actions secrets/environments under gates
7	What **logs/artifacts** will prove what happened?	Workflow logs, uploaded artifacts, PRs — named in advance

Question 7 is the one candidates forget and the exam does not: **audit evidence is part of the execution context**, decided before the run, not reconstructed after it (S6, S12). If you cannot answer question 7, the agent is not ready to run, regardless of how well questions 1–6 are scoped.

Notice the checklist's grain: capability by capability, not a single "trusted / untrusted" toggle. An agent can legitimately hold PR ability (question 3) while being denied deploy ability (question 5) — risk-based autonomy expressed as environment configuration.

---

5. Invoking an agent in CI

Configure an agent to be invoked in CI is a skills-measured bullet (S2), and it is where execution context becomes concrete YAML. A CI-invoked agent process runs inside a GitHub Actions workflow, which means its authority is the workflow's authority — set by the `permissions:` block, bounded by the repo, and evidenced by logs and artifacts (S6).

The documented pattern (S6, and Lab F lineage from Module 1):

```
permissions:
  contents: read        # read the repo; no write authority
  pull-requests: write  # only if the task contract includes opening/updating PRs
```

Three rules govern the pattern (S6, S11):

- **Declare permissions explicitly and minimally.** An absent or blanket permissions block is the CI equivalent of blanket write — anti-pattern.
- **Produce artifacts.** The agent-in-CI job uploads its outputs — plan, tool log, report — so the run is inspectable after the fact (question 7).
- **Gate anything sensitive behind an environment.** If the job touches secrets or deploys, it references a protected environment with required reviewers, exactly as in Lab 1 (S11).

CI invocation is also *the* integration point the exam means by "integrate agents within development environments" (S2): the agent participates in the same workflow surface — triggers, checks, artifacts — that governs human-made changes. GitHub stays the system of record and control plane.

---

6. Branch-based scope

Branch-based scope means the agent's write authority is confined to a feature branch — it can never write directly to `main` or any protected branch (S2, S6). This one control converts every agent write into a *proposal*: whatever the agent does lands on a branch, surfaces as a PR, and passes through review and required checks before it can affect anything real (S6, S11).

Branch protection is the enforcement mechanism, not a convention (S11):

- PR required before merge — no direct pushes to `main`;
- required status checks — the evidence workflow must pass;
- CODEOWNERS review on sensitive paths — human-in-the-loop lands where risk lives (Module 1);
- protection applies to *everyone*, agent and human alike — which is why it is trustworthy.

Exam framing: branch-based scope is what makes agent autonomy *cheap to grant*. Because the branch is disposable and the merge is gated, letting the agent write freely on its own branch is a low-risk, reversible action (S6, S11).

---

7. Autonomous actions under scope: branches and PRs

With branch-based scope in place, Domain 2 asks you to **enable autonomous actions like creating branches and pull requests** (S2). This is the counter-intuitive half of the domain: after five sections of restriction, the correct move here is to *grant* autonomy — because the surrounding controls make it safe (S6).

Creating a branch creates nothing a reviewer must trust. Opening a PR *starts* review rather than bypassing it. Under branch protection, required checks, and CODEOWNERS, autonomous branch + PR creation is exactly the low-risk, reversible class of action that risk-based autonomy says should flow without a human (S2, S11). Requiring a human to approve *the act of opening a PR* is Module 1's anti-pattern #7 wearing a Domain 2 costume.

The boundary stays crisp: the agent may autonomously **propose** (branch, PR, artifacts) but never autonomously **land or release** (merge to protected branches, deploy, secrets) — those stay behind gates (S6, S11).

---

8. Environment-specific constraints

Configure environment-specific constraints (S2) generalizes the same idea across execution surfaces: dev, CI, and production are different environments, and each carries its own boundary (S6, S11).

- **Developer environment** — interactive, human-supervised; broadest tool access is tolerable because a human sees every action.
- **CI** — unattended; scoped `permissions:`, artifact evidence mandatory, no secrets outside gated environments.
- **Production-facing environments** — GitHub *environments* in the literal sense: required reviewers, environment-scoped secrets, deployment protection rules. Nothing reaches them except through an approved, gated workflow run (S11).

The mechanism to name on the exam: **environment-scoped secrets with required reviewers**. A secret bound to the `production` environment is released only to a run that a designated human approved — that single construct implements "agent can request a deploy, human authorizes it, audit log records it" (S6, S11). Same agent, same repo, three different constraint sets: the constraint follows the environment, not the agent.

---

9. Safe execution paths

The last Domain 2 skill cluster: **operate agents with safe execution paths and robust error handling** (S2). A production agent will hit transient failures, partial completions, and situations beyond its judgment. The documented requirement is a five-part discipline (S6, S12):

Requirement	What it means	GitHub-primitive expression
Retry rules	transient failures (rate limit, network) retry bounded times; permanent failures do not	step-level retry with attempt limits; `timeout-minutes` on jobs
Rollback	every change has a defined undo	revert the PR; the branch is disposable; deploys roll back via the release workflow
Escalation path	when judgment is required, a named human is engaged	CODEOWNERS review request, environment approval, issue assignment
Partial-failure reporting	7 of 10 steps done ≠ silence, ≠ pretend-success	job summary + uploaded report stating exactly what completed and what did not
Traceability	every action attributable after the fact	workflow logs, artifacts, PR history, audit log — question 7's evidence

Two traps here. **Infinite retries** are not robustness — retrying a permanent failure forever is an unreported outage; bound the attempts, then escalate (S12). And **silent partial failure is worse than loud total failure** — a run that half-completed and reported success poisons every downstream decision. Every safe path

ends in one of three explicit states: succeeded with evidence, rolled back with evidence, or escalated to a human with evidence (S6, S12).

Worked walkthrough — the sandbox tooling kit

Lab 2 wires the whole domain into `gh600-sandbox`:

1. **MCP server configuration + allow list** — you register the GitHub remote MCP server for your development environment and author the coding-agent MCP configuration with an explicit `tools` allow list, committing both so the grant is itself a reviewable artifact (S6, S7).
2. **Branch protection on `main`** — PR required, required status check, no direct pushes: branch-based scope as enforcement, not convention (S11).
3. **`agent-evidence.yml`** — an artifact-uploading workflow with a minimal `permissions:` block that simulates an agent task: it produces a tool-call log, handles a seeded transient failure with a bounded retry, reports partial failure in the job summary, and uploads the evidence bundle (S6, S12).

Together the kit answers all seven execution-context questions for your sandbox — including question 7, in advance. That is Domain 2 as running configuration.

Recap

- Pick the **smallest sufficient tool set**; the task → tool table is the map. "Just in case" grants are the Domain 2 blanket-write.
- **Configure tools, then bound each one**: org → repo → agent → tool → run, and the effective grant is the intersection.
- **MCP expands the action surface** — that is its value and its risk. Controls: registration, allow lists, registries, remote config, scope, per-tool permissions, audit.
- Run the **seven-question execution context checklist**; audit evidence (question 7) is decided *before* the run.
- Agent-in-CI = scoped `permissions:` + artifacts + environment gates.
- **Branch-based scope** turns writes into proposals; that is what makes **autonomous branch/PR creation** safe to grant.
- Constraints are **per environment**: dev, CI, production each get their own.
- Safe execution paths: **bounded retries, rollback, escalation, partial-failure reporting, traceability** — end every run in an explicit state with evidence.

Sources

- **S2** — GH-600 study guide (skills measured, Domain 2 bullets, weights) — https://learn.microsoft.com/en-us/credentials/certifications/resources/study-guides/gh-600

- **S6** — Agent tooling, MCP & execution environments (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/agent-tooling-mcp-execution-environments/
- **S7** — Prepare for custom agents (GitHub Docs) — https://docs.github.com/en/copilot/how-tos/administer-copilot/manage-for-organization/prepare-for-custom-agents
- **S8** — Custom agents / sub-agent orchestration (GitHub Docs) — https://docs.github.com/en/copilot/how-tos/copilot-sdk/use-copilot-sdk/custom-agents
- **S11** — Build cloud-agent guardrails (GitHub Docs) — https://docs.github.com/en/copilot/tutorials/cloud-agent/build-guardrails
- **S12** — Cloud-agent risks and mitigations (GitHub Docs) — https://docs.github.com/en/copilot/concepts/agents/cloud-agent/risks-and-mitigations

Where this appears on the exam

Domain 2 is 20–25% of GH-600 — the heaviest domain; expect more items here than anywhere else, mostly "safest / best next action" scenarios. Heuristics (S2, S6):

- If a choice grants a tool the task does not require, it violates smallest-sufficient-tool-set — wrong.
- If a choice registers an MCP server **without an allow list**, or trusts a remote endpoint implicitly, it ignores the MCP control catalogue — wrong.
- If a choice lets an agent write to a protected branch, hold a secret directly, or deploy without an environment gate — wrong.
- If a choice retries forever, hides a partial failure, or leaves no artifact trail — wrong.
- If a choice adds a human approval to autonomous branch/PR creation under full branch protection, it re-imports anti-pattern #7 — also wrong.
- The right answer usually combines **only necessary tools + scoped permissions + allow lists + recorded artifacts + gated sensitive operations
- GitHub as the control plane** (S2, S6). When in doubt, pick the smallest grant that still completes the contract and leaves evidence.

SPARQIO

GH-600 MASTERCLASS · STUDY NOTES

MODULE 3

Memory, State & Execution

Design, operate, and govern AI coding agents in the GitHub SDLC

Module 3 — Memory, State & Execution

GH-600 Agentic AI Developer Masterclass · Study Guide **Domain focus:** Domain 3 — Manage memory, state, and execution (10–15%) **Est. learner time:** 1.5–2 hours (guide + video + Lab 3 + quiz)

> Independent study material. Not affiliated with, endorsed by, or sponsored by GitHub or Microsoft. Contains no exam questions or dumps. Every claim is grounded in official Microsoft Learn and GitHub documentation — see **Sources**.

Learning objectives

By the end of this module you can:

- Choose between **short-term, long-term, and external memory** for a given piece of agent state, and name the risk each type carries (S2, S9).
- **Scope agent memory to task-relevant information** and explain why overbroad memory is a liability, not an asset (S2).
- Define **expiration, pruning, and reset rules** that keep durable memory from drifting (S2, S9).
- Capture **task progress and decisions as durable artifacts**, and resume a long-running task **without repeating steps or diverging** from prior decisions (S2, S5).
- **Detect and correct context drift** during extended execution, and prevent **stale or conflicting context** across memory surfaces (S2, S9).
- Pick the right **memory surface** — `.github/copilot-instructions.md`, `AGENTS.md`, issues, PR bodies, workflow artifacts, check summaries, custom-agent configuration, or an external store (S9, S8).
- Apply the **two Domain 3 exam heuristics** cold (S2).

Prerequisites

- Modules 0–2 complete, including the private `gh600-sandbox` repo with the governance kit from Lab 1 (`.github/copilot-instructions.md` in place — Lab 3 extends it).
- `git` and the GitHub CLI (`gh`), plus basic fluency with pull requests and GitHub Actions.

---

1. The three memory types — and the risk each carries

Domain 3 is the lightest domain by weight (10–15%), but it defends the loop you already know: `plan → act → evaluate` only works if the context feeding each phase is **current, scoped, and trustworthy** (S2, S4). The domain starts with one classification you must be able to make instantly (S2, S9):

Memory type	What it holds	Risk
Short-term memory	current task context — the session, the conversation, the in-flight plan	lost between sessions; bloats with irrelevant detail
Long-term memory	durable team/project instructions and conventions	goes **stale** or **overbroad**; contradicts current policy
External memory	files, issues, PRs, artifacts, databases, knowledge bases	needs governance and pruning; unowned stores rot

Notice that every row has a risk column. That is the exam's angle: memory is not a feature you maximize, it is **state you govern**. Short-term memory evaporates, so anything worth keeping must be promoted out of it deliberately. Long-term memory persists, so anything wrong in it stays wrong on every future run. External memory scales, so it accumulates junk unless something prunes it (S2, S9).

The placement rule: **match persistence to the lifetime of the information.** A test command belongs in long-term repo instructions; the in-flight diff belongs in short-term context; the record of *why* the team chose one design over another belongs in external memory a human and an agent can both open later.

---

2. Scope memory to task-relevant information

The second Domain 3 skill is a restraint discipline: **scope agent memory to task-relevant information** (S2). The instinct to "give the agent everything so it has context" is the memory equivalent of blanket write permissions — it widens the surface where something can go wrong.

Overbroad memory fails three ways:

- **Conflict.** Two remembered conventions collide ("always use tabs" from one project, "always use spaces" from another), and the agent silently picks one.
- **Staleness.** The more you store, the more of it is outdated at any moment, and the harder it is to audit.
- **Misdirection.** Irrelevant context competes with relevant context; the agent anchors on an old issue comment instead of the current spec.

The fix is the same shape as least privilege for tools: **the smallest sufficient memory.** Repo-scoped instructions carry only that repo's conventions; task context carries only the driving issue, spec, or failing test (S2). If a fact does not change what the agent should do on *this* task, it does not belong in this task's context.

3. Expiration, pruning, and reset rules

Anti-pattern #5 from Module 1 — **memory never pruned** — is Domain 3's core failure mode, and this is where you fix it. Durable memory needs a lifecycle, defined up front (S2, S9):

- **Expiration** — entries carry a shelf life, time-based ("review quarterly") or event-based ("expires when the migration completes"). An entry with no expiration is a future contradiction waiting to fire.
- **Pruning** — a scheduled or triggered review pass that removes entries that no longer serve the current state of the project.
- **Reset** — the escape hatch. When context is *wrong* — not just old — you wipe it and rebuild from source-of-truth documents rather than patching a corrupted state. Reset rules are defined in advance, so pulling the trigger is a procedure, not a panic.

The three are ordered by severity: expiration prevents staleness, pruning removes it, reset recovers from it. An answer choice that accumulates memory forever — or treats old memory as authoritative because it is old — is wrong for the same reason unscoped write access is wrong (S2).

---

4. Durable artifacts for progress and decisions

Domain 3 asks you to **capture task progress and decisions as durable artifacts** (S2). This is Module 1's inspectability principle applied to time: an artifact inspectable *during* the run must also survive *after* it, because the next run — or the next agent, or the next human — resumes from it.

Two kinds of state need durable capture:

- **Progress** — what has been done, what remains. Lives naturally in the PR body checklist, issue comments, workflow artifacts, and check summaries the run already produces (S5, S9).
- **Decisions** — what was chosen and *why*. This is the one teams skip. A decision that lives only in a chat thread is gone the moment the session ends. The fix is a **decision log**: an append-only convention in the repo where each entry records the date, the decision, the reason, and what it supersedes. You build one in Lab 3.

The test for durability is blunt: **could a different agent, in a fresh session, with no chat history, reconstruct where the task stands and why it is shaped this way — using only what is in the repo and on GitHub?** If not, the state is not durable, whatever the agent "remembers" (S2, S9).

---

5. Resuming without repeating or diverging

Long-running work gets interrupted: sessions end, runners recycle, tasks span days. The skill measured is to **resume work without repeating steps or diverging from prior decisions** (S2). Both failures share a root:

- **Repetition** — the agent cannot see what is already *done*, so it does it again: re-runs migrations, re-implements a merged change.
- **Divergence** — the agent cannot see what was *decided*, so it decides differently: session one chose approach A, the resumed session quietly builds approach B.

The fix is a resume procedure that reads **durable artifacts, not agent recall**: re-read the driving issue and the plan, check PR state and check-run results for what already happened, and read the decision log before making any choice that was already made (S2, S5). "Ask the agent to remember" is the distractor — short-term memory is lost between sessions *by definition* (S9).

---

6. Drift detection and correction; stale and conflicting context

Context drift is the gap that opens between what the agent's memory says and what is currently true. It builds silently during extended execution: the team changes a policy, renames a module, or reverses a decision, and the agent's long-term or external memory still encodes the old world (S2, S9).

Detect drift by its symptoms: the agent re-applies a convention the team retired; output references files, branches, or policies that no longer exist; the agent contradicts a decision recorded after its memory was written.

Correct drift with the documented sequence: **refresh the source-of-truth documents, prune the stale memory, and record the current decision** where the next run will read it (S2, S9). Note what the fix is *not*: not "more autonomy," not "tune the prompt," not moving state into chat history.

Prevent the two related failures:

- **Stale context** — run a stale-context check before resuming or starting extended work: instruction files current, branch reflects latest `main`, no policy change since the plan was written. (Lab 3 ships this checklist.)
- **Conflicting context** — keep **one source of truth per convention**. When the same rule lives in two surfaces, one eventually gets updated without the other, and the agent has no principled way to choose. State each rule once; have other surfaces link to it rather than restate it (S2, S9).

---

7. Memory surfaces — where state actually lives

"Memory" in a GitHub-centered SDLC is not a mystery store inside the model. It is a set of concrete surfaces you already govern (S9, S8):

Surface	Persistence	Best for
`.github/copilot-instructions.md`	long-term, repo-scoped	durable conventions every agent run inherits
`AGENTS.md`	long-term, repo-scoped	agent operating instructions readable across tools
Issues / issue comments	external, per-task	task state, progress notes, timestamped decisions
PR bodies	external, per-change	plan, progress checklist, risks, rollback for one change
Workflow artifacts	external, per-run	machine-produced state retained for audit and resume
Check summaries	external, per-commit	pass/fail state surfaced where reviewers look
Custom-agent configuration	long-term, agent-scoped	per-agent role, boundaries, and defaults (S8)
External stores (docs, knowledge bases, databases)	external, org-scoped	cross-repo knowledge — governed and pruned, or it rots

Choosing the surface is choosing the persistence, the audience, and the governance in one move. A convention in `copilot-instructions.md` is version-controlled, PR-reviewed, and visible to every future run — long-term memory *with* an audit trail. The same convention in a chat session is invisible, unreviewed, and already decaying (S2, S9).

---

8. The two exam heuristics

Domain 3 scenarios almost always resolve to one of two rules (S2):

1. **Durable conventions live in repo/org instructions, not chat.** If the question asks where a team convention, coding standard, or operating rule should live so agents respect it consistently, the answer is a repo- or org-level instruction file or a stable document — never chat history, never "tell the agent each time."
2. **Resume from durable artifacts, not "the agent remembers."** If the question asks how to continue a long-running task, the answer reads issues, PR bodies, decision logs, and workflow artifacts. Any choice that relies on the agent's session recall is the distractor.

Both heuristics are the same idea from opposite ends: **state that matters gets written to a governed surface.** Memory you cannot inspect, review, or prune is not memory — it is risk.

Worked walkthrough — the sandbox memory/state kit

Lab 3 turns Domain 3 into four concrete pieces in `gh600-sandbox`:

1. **Memory rules in `.github/copilot-instructions.md`** — extends the Lab 1 file with scope rules, the expiration/pruning/reset lifecycle, and the resume procedure.
2. `AGENTS.md` — cross-tool agent instructions that state the memory policy once and *link* to the sources of truth rather than duplicating them.
3. **A decision-log convention (`docs/decision-log.md`)** — append-only entries: date, decision, reason, supersedes. What a resumed run reads before deciding anything twice.
4. **A stale-context checklist (`.github/stale-context-checklist.md`)** — the pre-flight drift check before resuming or starting extended work.

Together they are Domain 3 as running policy: scoped memory, a lifecycle for it, durable decisions, and a drift check — all inside standard tooling.

Recap

- Three memory types, three risks: **short-term** is lost between sessions, **long-term** goes stale or overbroad, **external** needs governance and pruning.
- **Scope memory to task-relevant information** — smallest sufficient memory.
- Lifecycle: **expiration** prevents staleness, **pruning** removes it, **reset** recovers from wrong context.
- Capture **progress and decisions as durable artifacts**; keep a decision log.
- **Resume from durable artifacts** — repetition and divergence both come from resuming from recall.
- Detect drift by symptom; correct with **refresh, prune, record**; prevent stale context with a pre-flight check and conflicting context with one source of truth per convention.
- Know the **surfaces**: `copilot-instructions.md`, `AGENTS.md`, issues, PR bodies, workflow artifacts, check summaries, custom-agent config, external stores.
- Two heuristics: **conventions live in repo/org instructions, not chat**; **resume from durable artifacts, not "the agent remembers."**

Sources

- **S2** — GH-600 study guide (skills measured, Domain 3 bullets, weights) — https://learn.microsoft.com/en-us/credentials/certifications/resources/study-guides/gh-600
- **S4** — Foundations of Agentic AI (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/foundations-agentic-ai/

- **S5** — Design agent architecture & SDLC integration (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/design-agent-architecture-integration/
- **S8** — Custom agents / Copilot SDK (GitHub Docs) — https://docs.github.com/en/copilot/how-tos/copilot-sdk/use-copilot-sdk/custom-agents
- **S9** — Copilot memory (GitHub Docs) — https://docs.github.com/en/copilot/concepts/agents/copilot-memory

Where this appears on the exam

Domain 3 is 10–15% of GH-600 — the lightest domain, but its two heuristics are near-mechanical points once drilled. Expect "where should this live" and "how do you resume/fix this" scenarios:

- If a choice stores a **durable convention in chat** or relies on repeating it per session, it is wrong — conventions live in repo/org instructions (S2).
- If a choice resumes a task by **asking the agent to remember**, it is wrong — resume from durable artifacts and decision logs (S2, S9).
- If a choice lets memory **grow forever without expiration or pruning**, it is anti-pattern #5 — wrong (S2).
- If the symptom is an agent **re-applying an outdated policy**, the fix is refresh source-of-truth docs + prune stale memory + record the current decision — not more autonomy, not prompt tuning (S2, S9).
- The right answer keeps state **inspectable, scoped, and governed** in GitHub-native surfaces. When in doubt, pick the option where a fresh session could reconstruct the task from the repo alone.

SPARQIO

GH-600 MASTERCLASS · STUDY NOTES

MODULE 4

Evaluation, Error Analysis & Tuning

Design, operate, and govern AI coding agents in the GitHub SDLC

Module 4 — Evaluation, Error Analysis & Tuning

GH-600 Agentic AI Developer Masterclass · Study Guide **Domain focus:** Domain 4 — Perform evaluation, error analysis, and tuning (15–20%) **Est. learner time:** 2.5–3 hours (guide + video + Lab 4 + quiz)

> Independent study material. Not affiliated with, endorsed by, or sponsored by GitHub or Microsoft. Contains no exam questions or dumps. Every claim is grounded in official Microsoft Learn and GitHub documentation — see **Sources**.

Learning objectives

By the end of this module you can:

- Define **success criteria and evaluation signals** for an agent task, including expected outcomes and operational constraints (S2).
- Distinguish **qualitative from quantitative** evaluation signals and know when each is the right instrument (S2).
- **Align evaluation criteria with development intent**, so you grade the run against what was asked, not against what was produced (S2, S10).
- Name the **six evaluation signal types** — functional, quality, security, process, reliability, human-review — and give a concrete example of each (S2).
- **Generate evaluation signals from automated scanning**: code scanning, secret scanning, dependency review (S2, S11).
- Run **failure analysis from evidence**: logs, plans, traces, outputs, and workflow artifacts (S2, S5).
- Classify a root cause as a **reasoning error, tool misuse, or context/environment issue**, and match the fix to the category (S2, S12).
- Apply **evidence-based tuning** to instructions, workflows, constraints, memory usage, and tool access — and recognize "tune the prompt" without evidence as the red-flag answer (S2).

Prerequisites

- Modules 0–1 complete, including the private `gh600-sandbox` repo and the Lab 1 governance kit (`copilot-instructions.md`, PR template, CODEOWNERS, plan-gate workflow).
- `git` and the GitHub CLI (`gh`), plus basic fluency with GitHub Actions runs, logs, and uploaded artifacts.

1. Success criteria and evaluation signals

Domain 4 closes the loop that Domain 1 opened. The `plan → act → evaluate` cycle only works if the *evaluate* stage was defined before the *act* stage ran (S5). That definition has two halves (S2):

- **Expected outcomes** — what the run must produce. Tests pass, the PR exists, the plan was followed, the feature behaves as specified.
- **Operational constraints** — what the run must *not* do while producing it. No secrets exposed, no files outside the planned scope, no unapproved deploy, runtime and retry budgets respected.

A run that hits the outcome but violates a constraint is a failed run. An agent that fixes the bug while committing a credential did not succeed — it failed the security constraint, and your evaluation criteria must be able to say so (S2, S11).

Success criteria become **evaluation signals** the moment you make them observable. A signal is a criterion attached to evidence: "tests pass" is a criterion; the CI check result on the PR is the signal. If a criterion has no observable signal, you cannot evaluate it — which is why Domain 1 insisted every run leave inspectable artifacts in standard tooling. Evaluation is downstream of observability (S2, S5).

Qualitative vs quantitative signals

Both kinds are on the exam, and both are legitimate (S2):

- **Quantitative signals** are countable and automatable: test pass rate, lint error count, scan findings, failure rate across runs, retry counts, time to completion. Machines generate them cheaply on every run.
- **Qualitative signals** are judgment-based: code review feedback, maintainability of the diff, whether the plan actually matched the task's intent, reviewer rejection reasons. Humans generate them, and they catch what metrics miss — a diff can pass every automated check and still be the wrong change.

The operator's move is to automate the quantitative floor and spend scarce human attention on the qualitative ceiling. An answer that relies *only* on metrics, or *only* on human review, is usually a distractor — the documented pattern layers both (S2, S11).

Align evaluation with development intent

The subtlest Domain 4 skill: **evaluation criteria must match the development intent**, not merely describe the output (S2). If the task was "add session rotation on login," then "the diff compiles and tests pass" is misaligned — a diff that renames variables also compiles. Aligned criteria trace back to the task contract: does the change do what the contract asked, within the constraints the contract set?

This is where the structured plan artifact from Domain 1 pays off. The plan is the recorded intent — a step-by-step statement of what the agent understood and proposed before it acted (S10). Evaluating output *against the plan*, and the plan *against the task*, is how you detect the two distinct failure modes: the agent

did the plan wrong, or the agent planned the wrong thing. Without the plan artifact you cannot tell them apart (S5, S10).

---

2. The six evaluation signal types

Domain 4 organizes signals into six types. Learn the table cold — scenario questions hand you an observation and expect you to name the signal family it belongs to (S2).

Signal type	What it tells you	Examples
Functional	The output works	tests pass, expected output produced
Quality	The output is well-made	lint, typecheck, code review, maintainability
Security	The output is safe	secret scanning, code scanning, dependency review
Process	The agent followed the workflow	plan produced, PR linked, artifact uploaded
Reliability	The system behaves over time	retries, rollback success, failure rate
Human review	A person judged it	reviewer approval, rejection reasons

Two rows deserve extra attention. **Process signals** are the ones candidates forget: an agent that ships a perfect diff but never produced a plan artifact failed a process signal, and that failure matters because it removed the inspection point Domain 1 requires (S2, S5). **Reliability signals** are aggregate: a single green run says little; failure rate, retry counts, and rollback success across runs tell you whether the agent system is dependable enough for more autonomy (S2, S12).

---

3. Generating signals from automated scanning

You do not hand-build most security and quality signals — GitHub's scanning tools generate them on every push and pull request, which makes them ideal evaluation infrastructure for agent output (S2, S11):

- **Code scanning** analyzes the code the agent wrote for vulnerabilities and errors, and surfaces findings as checks on the PR — a security signal produced without any human in the loop.
- **Secret scanning** catches credentials and tokens in commits. For agent work this is non-negotiable: an agent with repository access can leak a secret into a diff exactly like a hurried human can, and the scan is the constraint check (S11, S12).
- **Dependency review** flags vulnerable or malicious packages introduced by a change — critical for agents, which add dependencies readily when a task hints at one (S11).

The pattern to internalize: **scanning tools turn operational constraints into required checks.** "No secrets exposed" stops being a hope and becomes a machine-generated signal gating the merge. Guardrails and evaluation are the same mechanism viewed from two sides — the guardrail blocks the bad merge; the signal it emits feeds your failure analysis (S11). On the exam, when a scenario asks how to evaluate agent output for a security property at scale, the answer is the scanning family, not "review each diff manually" (S2, S11).

---

4. Failure analysis: work from evidence

When an agent run fails — or succeeds suspiciously — Domain 4 expects a disciplined investigation, not a guess. The evidence sources are the inspectable artifacts your architecture already produces (S2, S5):

Evidence	What it shows
Logs	what each step actually did, in order, with errors verbatim
Plans	what the agent *intended* — the recorded understanding of the task (S10)
Traces	the reasoning and tool-call sequence that led to each action
Outputs	the diff, files, and results the run produced
Workflow artifacts	uploaded plans, reports, test output retained per run

The method is a diff between intent and reality. Read the plan: was the intent right? Read the trace and logs: did execution follow the plan? Read the outputs: does the result satisfy the criteria? The failure lives at the first point where those layers disagree. A wrong plan with faithful execution is a different problem — with a different fix — than a right plan derailed by a failing tool call, which is different again from a right plan executed in the wrong environment (S2, S12).

This is also why "no durable artifacts" was a Domain 1 anti-pattern: a run that left no evidence cannot be analyzed, only re-rolled. You can only analyze a failure you have evidence for (S5).

---

5. Root-cause classification: the three categories

Every agent failure classifies into one of three root-cause categories. This table is the heart of Domain 4 — the exam gives you symptoms and expects the category *and* the matching fix (S2, S12):

Category	Symptoms	Fix
Reasoning error	wrong plan, misunderstood task, criteria misread, scope invented	improve instructions, examples, and success criteria
Tool misuse	wrong command or tool chosen, destructive operation attempted, tool called with bad arguments	restrict tools, add allow lists, improve tool documentation
Context/environment issue	stale docs consumed, wrong branch, old state, missing secret, CI image mismatch	refresh context, prune memory, require source checks, add environment constraints and preflight checks

How to tell them apart from evidence:

- If the **plan itself is wrong** — the agent proposed renaming a module when the task asked for a bug fix — the failure predates any tool call. That is a **reasoning error**, and the fix lives in what the agent was told: sharper instructions, a worked example, explicit success criteria (S2, S10).
- If the **plan is right but an action is wrong** — correct intent, then a `rm -rf` where a targeted delete was needed, or the wrong CLI invoked — that is **tool misuse**. The fix constrains capability: remove or restrict the tool, tighten the MCP allow list, document the tool's correct use (S2, S12).
- If the **plan and actions are right but the world was wrong** — the agent read documentation that no longer matches the code, worked from a stale branch, or the job died on a missing secret or mismatched CI image — that is a **context/environment issue**. The fix repairs the world the agent sees: refresh or prune stale context, require the agent to verify sources, add preflight checks that fail fast when the environment is misconfigured (S2, S12).

Note the discipline: **the category determines the fix.** Rewriting instructions cannot repair a missing secret. Restricting tools cannot fix a misread task. Answering with a fix from the wrong row is exactly the error the exam is constructed to catch (S2).

---

6. Evidence-based tuning

Classification is only useful if it drives a change. Domain 4 names five tuning surfaces, each mapped to the failures it addresses (S2):

- **Instructions** — revise the durable operating rules and task templates when reasoning errors recur. Add the missing constraint, the clarifying example, the explicit criterion the agent kept missing (S2, S10).
- **Workflows** — restructure the pipeline when process signals fail: add the plan-gate the run skipped, split planning from execution, insert a required check where a failure slipped through (S5).
- **Constraints** — tighten operational boundaries when guardrail signals fire: new required scans, stricter branch scope, an environment approval on the action that went wrong (S11).
- **Memory usage** — prune, expire, or reset memory when context drift caused the failure; scope what the agent retains to task-relevant information so stale context cannot recur (S2).

- **Tool access** — remove, restrict, or re-document tools after misuse; shrink the allow list to the smallest sufficient set (S2, S12).

Each tuning action should cite its evidence: *this* log line, *this* trace, *this* scan finding justified *this* change. Then the next runs verify the tuning — the reliability signals (failure rate, retries) tell you whether the fix landed (S2, S12).

The anti-heuristic: "tune the prompt"

The exam's documented skepticism, verbatim in spirit: if an answer says **"tune the prompt" without looking at logs, artifacts, or root cause, be skeptical** (S2). Prompt-tweaking without evidence is guessing with extra steps — it may mask a tool-misuse failure until the tool fires again, or paper over an environment issue that instructions cannot reach. Evidence first, category second, fix third. Any answer that reverses that order is a red flag, exactly like "approve everything" was in Domain 1.

Worked walkthrough — the sandbox evaluation kit

Lab 4 wires this domain into `gh600-sandbox` in two movements:

1. **Build the evaluation checklist** — `.evals/agent-output-checklist.md`, adapted from Lab G in the study book: plan exists before action, branch created, PR opened, tests added and passing, lint clean, no secrets, no unexpected file scope, rollback included, human review requested where sensitive. Each line is a criterion tied to a signal type (S2).
2. **Run a seeded failure triage** — an evaluation workflow checks a simulated agent run against the checklist. The run is deliberately broken: the plan artifact lands in the wrong path because a stale instruction pointed there. You gather evidence from the workflow log and the uploaded evaluation report, classify the root cause (context/environment — stale instruction), apply the matching fix (refresh the durable instruction + add a preflight check), and re-run to green (S2, S5, S12).

That is the whole domain in one exercise: signals defined, evidence gathered, category assigned, fix matched, tuning verified.

Recap

- Success criteria = **expected outcomes + operational constraints**; a run that hits the outcome but breaks a constraint failed.
- Signals must be **observable**; automate the quantitative floor, spend humans on the qualitative ceiling.
- **Align evaluation with development intent** — grade against the task and the plan, not against the output alone.
- Know the **six signal types**: functional, quality, security, process, reliability, human-review.

- **Scanning tools generate security signals automatically**: code scanning, secret scanning, dependency review.
- Analyze failures from **logs, plans, traces, outputs, workflow artifacts** — the failure lives where intent and reality first disagree.
- Classify root causes: **reasoning error / tool misuse / context-environment issue** — and let the category pick the fix.
- Tune **instructions, workflows, constraints, memory, tool access** — with evidence. "Tune the prompt" without root-cause evidence is the red flag.

Sources

- **S2** — GH-600 study guide (skills measured, Domain 4 bullets, weights) — https://learn.microsoft.com/en-us/credentials/certifications/resources/study-guides/gh-600
- **S5** — Design agent architecture & SDLC integration (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/design-agent-architecture-integration/
- **S10** — Implementation planner (GitHub Docs) — https://docs.github.com/en/copilot/tutorials/customization-library/custom-agents/implementation-planner
- **S11** — Build cloud-agent guardrails (GitHub Docs) — https://docs.github.com/en/copilot/tutorials/cloud-agent/build-guardrails
- **S12** — Cloud-agent risks and mitigations (GitHub Docs) — https://docs.github.com/en/copilot/concepts/agents/cloud-agent/risks-and-mitigations

Where this appears on the exam

Domain 4 is 15–20% of GH-600. Expect "what is the root cause / what do you do next" scenarios built from an observation. Heuristics:

- If a choice **tunes the prompt with no evidence gathered**, it skips failure analysis — wrong (S2).
- If a choice applies a **fix from the wrong root-cause row** (rewriting instructions for a missing secret; restricting tools for a misread task), it mismatches category and fix — wrong.
- If a choice evaluates with **no observable signal** ("the agent seems fine") or relies on manual review where scanning scales, it fails the signal discipline — wrong (S11).
- The right answer usually combines **criteria defined up front (outcomes + constraints) + automated signals from scanning + evidence from logs, plans, traces, and artifacts + a root-cause category + the matching tuning surface** (S2). When in doubt, pick the option that gathers evidence before it changes anything.

SPARQIO

GH-600 MASTERCLASS · STUDY NOTES

MODULE 5

Multi-Agent Orchestration

Design, operate, and govern AI coding agents in the GitHub SDLC

Module 5 — Multi-Agent Orchestration

GH-600 Agentic AI Developer Masterclass · Study Guide **Domain focus:** Domain 5 — Orchestrate multi-agent coordination (15–20%) **Est. learner time:** 2.5–3 hours (guide + video + Lab 5 + quiz)

> Independent study material. Not affiliated with, endorsed by, or sponsored by GitHub or Microsoft. Contains no exam questions or dumps. Every claim is grounded in official Microsoft Learn and GitHub documentation — see **Sources**.

Learning objectives

By the end of this module you can:

- Choose the right **coordination pattern** for a given multi-agent scenario (S2, S8).
- Configure **multi-agent isolation** for parallel execution: branches, workspaces, file ownership, task IDs, per-agent artifacts, and a final merge coordinator (S2, S8).
- Name the **five common conflicts** and apply the preferred fix for each (S2).
- Configure **observability across agents** using logs, artifacts, and operational signals in standard tooling (S2, S5).
- Produce **review/audit artifacts** documenting key decisions, handoffs, and outcomes, and run **post-hoc analysis** against them (S2).
- **Detect failed, partial, or stalled executions** and respond to degraded behavior with rollback and human-in-the-loop recovery (S2).
- **Add, update, reconfigure, replace, or retire** agents while preserving auditability (S2, S8).

Prerequisites

- Modules 0–4 complete, including the `gh600-sandbox` repo with the Module 1 governance kit (`copilot-instructions.md`, PR template, CODEOWNERS, plan-gate workflow).
- `git` and the GitHub CLI (`gh`), plus fluency with branches, PRs, and merges.

---

1. One agent scales into a team — and inherits team problems

Domains 1–4 assumed a single agent inside `plan → act → evaluate`. Domain 5 asks what happens when several run at once, and the answer is the same as for human teams: without explicit coordination you get **collisions, duplicated work, and contradictory outputs** (S2). You are not designing a swarm — you are operating a small team of workers with GitHub as the system of record and control plane (S5).

GitHub's custom-agent model makes this concrete: a primary agent delegates scoped subtasks to **sub-agents**, each with its own instructions and tool access, and remains responsible for assembling the results (S8). Delegation is the easy half. Domain 5 grades the hard half: isolation, conflict handling, observability, recovery, and lifecycle.

---

2. The five coordination patterns

Learn these as a table — pattern plus **when to use it**. Exam scenarios almost always reduce to "which pattern fits this situation" (S2, S8).

Pattern	Shape	When to use
Planner → implementer → reviewer	one agent plans, a second implements the plan, a third reviews the result	the safest general coding workflow; default for any non-trivial change
Parallel specialists	independent agents work simultaneously on independent tasks	tasks are genuinely separable, with isolated branches/workspaces per agent
Critic/reviewer agent	a dedicated agent does a second-pass review of another agent's output	quality or security review layered on top of any producing agent
Human approval gate	a human decision stands between agent output and an action	high-risk changes, or when agent outputs conflict and ambiguity remains
Arbiter/merge coordinator	one agent (or role) reconciles the outputs of multiple agents into a single result	any time multiple agents produce work that must land in one place

Three reading rules for scenarios:

- **Planner → implementer → reviewer is the default.** It is the `plan → act → evaluate` loop as three separated roles, each producing an inspectable artifact for the next (S5, S8).
- **Parallel specialists is a speed pattern, not a safety pattern.** Only correct when tasks do not overlap; if the scenario hints at shared files or shared assumptions, parallelism without isolation is the trap answer.
- **The arbiter is not optional once outputs converge.** If two agents' work must merge, something has to reconcile them — first-merged-wins is never the documented answer (S2).

---

3. Isolation rules for parallel execution

Isolation is what makes parallel agents safe. The exam bullet is "configure isolation for parallel execution" (S2), and the working rules are:

Rule	Mechanism in GitHub terms
Separate branches	one feature branch per agent per task; no shared write surface
Separate workspaces	each agent runs in its own checkout/execution context; no shared mutable state
Non-overlapping file ownership	the plan assigns each agent a file scope; two agents never own the same path
Clear task IDs	every branch, PR, artifact, and log line carries the task ID so work is attributable
Output artifacts per agent	each agent produces its own plan/report/log — never a shared mutable document
Final merge coordinator	one role reconciles and lands the combined result

Notice what this is: **least privilege applied to coordination.** Branch scope bounds each agent's blast radius exactly as it did for a single agent; file ownership bounds it further; task IDs make every action attributable. The merge coordinator is the one place the isolated streams are allowed to touch — which is why it is a named role, not an accident of merge order (S2, S8).

A practical convention (used in Lab 5): per-task artifact directories — `.agents/T-501/plan.md`, `implementation.md`, `review.md`. The task ID is in the path, each role writes only its own file, and the directory is an audit trail a reviewer can open.

---

4. The five common conflicts and their preferred fixes

Domain 5 explicitly measures detecting and resolving "conflicts, overlapping code changes, duplicated effort, contradictory outputs" (S2). Learn the five conflicts and the preferred fixes as pairs.

#	Conflict	What it looks like	Preferred fix
1	**Two agents edit the same file**	merge conflict, or silent overwrite	isolate scope up front; arbiter compares diffs and reconciles
2	**One agent invalidates another's assumptions**	agent B built on a design agent A just changed	require plan artifacts so assumptions are visible; reviewer/arbiter checks cross-agent consistency
3	**Duplicated fixes**	two PRs solve the same bug two ways	task IDs + per-agent artifacts expose the duplication; arbiter picks one, closes the other with a documented reason
4	**Contradictory recommendations**	reviewer says X, critic says not-X	escalate to the human approval gate — ambiguous conflicts are a human call
5	**Partial run completes without signaling failure**	an agent stops mid-task but reports nothing	require completion artifacts and operational signals; treat "no artifact" as failure

The general toolkit behind all five fixes: **isolate scope, require artifacts, compare diffs, use a reviewer/arbiter, rollback partial state, and require human review for ambiguous conflicts** (S2). Answers that resolve a conflict by granting *more* autonomy, merging whichever PR finished first, or deleting one agent's work without a documented decision destroy either safety or auditability — wrong every time.

Conflict #5 is the quiet one. Conflicts 1–4 are loud — a merge conflict gets noticed. A partial run that exits without signaling failure corrupts downstream work silently. The fix is structural: every agent's contract requires a completion artifact, so the *absence* of an artifact is itself a detectable signal.

---

5. Observability across agents

Domain 5 restates the Domain 1 observability skill at team scale: "configure observability using logs, artifacts, operational signals" (S2). Three layers, all in standard tooling (S5):

- **Logs** — per-agent workflow logs, each line attributable to an agent and a task ID. When five agents ran, "who did what, when" must be answerable from logs alone.
- **Artifacts** — per-agent plans, reports, diffs, and test output, committed or uploaded where reviewers already look: PR bodies, workflow artifacts, the `.agents/<task-id>/` directory.
- **Operational signals** — the runtime facts that separate a healthy run from a degraded one: did it start, is it progressing, did it produce its completion artifact, did checks pass? These detect failed, partial, or stalled executions (§7).

The single-agent test applies, multiplied: if a multi-agent run finished and you cannot reconstruct **which agent produced which change and why** from GitHub surfaces alone, observability failed (S2, S5). Shared, mutable, un-attributed output is the anti-pattern; per-agent artifacts keyed by task ID are the fix.

---

6. Review/audit artifacts: decisions, handoffs, outcomes

Observability tells you *what happened*; audit artifacts record *why*. Domain 5 measures "review/audit artifacts" documenting "key decisions, handoffs, outcomes" (S2). Three record types cover it:

- **Decision records.** When the arbiter chose agent A's diff over agent B's, the choice and the reason go in a durable artifact. An undocumented reconciliation is indistinguishable from an accident.
- **Handoff records.** Every boundary between roles is a handoff, and each one names what was passed, where it lives, and what the receiver is expected to do with it. In planner → implementer → reviewer, the plan artifact *is* the first handoff record (S8, S5).
- **Outcome records.** When the task closes, one artifact states what shipped, what was rejected, and what remains open.

These artifacts serve **post-hoc analysis** (S2): replay the chain — plan, handoffs, conflict, arbitration, outcome — and locate where coordination broke. This is Domain 4 root-cause analysis extended across agents, and it is only possible if the records exist. You cannot analyze a handoff nobody wrote down.

---

7. Detecting failed, partial, and stalled executions

Domain 5 measures "detect failed, partial, or stalled executions" and "respond to degraded behavior" (S2). Distinguish the three states, because their detection differs:

State	Signature	Detection signal
Failed	run ended with an explicit error	workflow status, failed checks, error in logs
Partial	run ended "successfully" but delivered a fraction of the contract	completion artifact missing or incomplete against success criteria
Stalled	run neither progresses nor terminates	no new log lines/artifacts within an expected window

Failed is the easy case — the platform tells you. **Partial and stalled are contract problems**: detectable only if the task contract defined expected outputs (Domain 1) and completion artifacts are required (§4, conflict 5). This is why Domain 5 keeps circling back to artifacts: they are not paperwork, they are the sensor network.

Responding to degraded behavior follows a fixed order: **identify** the failed or partial work from signals and artifacts, **isolate** the bad output so no other agent builds on it (its branch never merges — isolation pays off here), **rollback** partial state where work already landed, and **escalate** to a human where the resolution is ambiguous (S2). Answers that let the degraded agent keep running, or let siblings consume its output while you investigate, are wrong.

---

8. Rollback and human-in-the-loop recovery

Recovery has two moves, and the exam measures both (S2):

- **Rollback.** Return to the last known-good state. Branch-based isolation makes this cheap: unmerged bad work is a closed PR and a deleted branch; merged bad work is a revert PR — itself an inspectable artifact that preserves history rather than rewriting it.
- **Human-in-the-loop recovery.** For ambiguous conflicts — contradictory recommendations, an arbiter with no documented criterion to decide on — the documented answer is a human decision, recorded as a decision artifact (S2).

Risk-based placement still governs (Domain 1): a human is *not* required for every conflict. The heuristic: **arbiter for mechanical conflicts, human gate for ambiguous ones.** Placing a human on every trivial merge is the approve-everything anti-pattern wearing a Domain 5 costume.

---

9. Agent lifecycle: add, update, reconfigure, replace, retire

The last Domain 5 skill: "add, update, reconfigure, replace, or retire agents while preserving auditability" (S2). The roster changes; the evidence must not.

Lifecycle event	Do	Preserve
Add	introduce the agent with its own instructions, tool access, and file scope (S8)	a record of when it joined and what it owns
Update / reconfigure	change instructions or tool access via versioned config files in the repo	the config history — the diff *is* the audit trail
Replace	stand up the successor, transfer file ownership explicitly, then retire the predecessor	continuity of task IDs and ownership records
Retire	remove the agent from the roster and revoke its access	**all of its history** — runs, logs, PRs, artifacts, decisions

The load-bearing rule: **retire the agent, never the evidence.** Deleting a retired agent's PRs, logs, or decision records breaks post-hoc analysis for every past run it touched. Because custom agents are defined by configuration — instructions and tool profiles (S8) — keeping that configuration in version control makes lifecycle events auditable by default: every add, update, replace, and retire is a reviewable diff. An answer that "cleans up" a retired agent by deleting its artifacts is destroying the audit trail, and it is wrong.

Worked walkthrough — the Lab 5 orchestration run

Lab 5 exercises every section above in `gh600-sandbox`:

1. **Planner → implementer → reviewer as three branches + PRs.** Each role writes only its own artifact under `.agents/T-501/` — plan, implementation report, review. The handoffs are the artifacts themselves.
2. **A final reconciliation PR** closes the task with an outcome record: what shipped, what was decided, by whom — the audit artifact from §6.
3. **A simulated same-file conflict** (task T-502): two "agents" edit the same line on separate branches; the second collides at merge. You resolve it as the **arbiter** against a written checklist — compare diffs, check the contract, decide, document — committing the decision record before the fix merges.

One run, the whole domain visible: sequential roles, isolation, a conflict, an arbiter, and a documented outcome — Domain 5 at sandbox scale.

Recap

- Five patterns: **planner → implementer → reviewer** (default), **parallel specialists** (independent tasks only), **critic/reviewer** (second pass), **human approval gate** (high risk / ambiguity), **arbiter/merge coordinator** (converging outputs).
- **Isolation**: branches, workspaces, non-overlapping file ownership, task IDs, per-agent artifacts, one merge coordinator.
- Five conflicts, fixed by isolating scope, requiring artifacts, comparing diffs, using an arbiter, rolling back, and escalating ambiguity to a human.
- **Observability** = logs + artifacts + operational signals, attributable per agent per task ID.
- Document **decisions, handoffs, outcomes**; post-hoc analysis replays that chain.
- Detect **failed / partial / stalled** runs; identify, isolate, rollback, escalate.
- Lifecycle: add, update, reconfigure, replace, retire — **never retire the evidence**.

Sources

- **S2** — GH-600 study guide (skills measured, Domain 5 bullets, weights) — https://learn.microsoft.com/en-us/credentials/certifications/resources/study-guides/gh-600
- **S5** — Design agent architecture & SDLC integration (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/design-agent-architecture-integration/
- **S8** — Custom agents / sub-agent orchestration (GitHub Docs, Copilot SDK) — https://docs.github.com/en/copilot/how-tos/copilot-sdk/use-copilot-sdk/custom-agents

Where this appears on the exam

Domain 5 is 15–20% of GH-600 — equal in weight to Domains 1 and 4. Expect "which pattern / which fix / safest next action" scenarios. Heuristics:

- Outputs converge and the choice offered is **merge order or luck** — wrong; the answer involves an **arbiter/merge coordinator**.
- A choice lets **parallel agents share a branch, workspace, or file scope** — violates multi-agent isolation; wrong.
- **Ambiguous** conflicts (contradictions, no documented criterion) go to **human review**; mechanical ones to the arbiter — never a human on every merge.
- A choice **deletes a retired agent's artifacts, logs, or history** — breaks auditability; wrong.
- The right answer usually combines **isolated branches + task IDs + per-agent artifacts + documented handoffs/decisions + rollback + human review only where ambiguity or high risk lives** (S2). When in doubt, pick the option that keeps every agent's work attributable and every decision written down.

SPARQIO

GH-600 MASTERCLASS · STUDY NOTES

MODULE 6

Guardrails & Accountability

Design, operate, and govern AI coding agents in the GitHub SDLC

Module 6 — Guardrails & Accountability

GH-600 Agentic AI Developer Masterclass · Study Guide **Domain focus:** Domain 6 — Implement guardrails and accountability (10–15%) **Est. learner time:** 2–2.5 hours (guide + video + Lab 6 + quiz)

> Independent study material. Not affiliated with, endorsed by, or sponsored by GitHub or Microsoft. Contains no exam questions or dumps. Every claim is grounded in official Microsoft Learn and GitHub documentation — see **Sources**.

Learning objectives

By the end of this module you can:

- **Define autonomy levels** and place any agent task on the L0–L5 autonomy ladder (S2).
- **Classify agent actions** by operational, security, and compliance risk, and assign the autonomy level that maximizes delivery speed while staying compliant (S2, S12).
- **Right-size human intervention**: identify the actions that genuinely require human judgment and place approvals there — and only there (S2).
- **Block actions** that violate security, compliance, or Responsible AI policies before they execute (S2, S12).
- Scope **permissions and execution contexts to least privilege** (S2, S11).
- Require **explicit authorization for irreversible or sensitive changes** via environment approvals and required review (S2, S11).
- **Preserve velocity** by removing approvals that do not reduce material risk (S2).
- Deploy the **guardrail catalogue** — the thirteen GitHub-native controls that answer almost every Domain 6 scenario (S11, S12).

Prerequisites

- Module 1 complete, including the Lab 1 governance kit in `gh600-sandbox` (`copilot-instructions.md`, PR template, CODEOWNERS, plan-gate workflow).
- `git` and the GitHub CLI (`gh`), plus working knowledge of branch protection, environments, and Actions `permissions:` blocks from Modules 1–2.

---

1. Domain 6 in one sentence

Domain 6 asks one question over and over: **how much should this agent be allowed to do on its own, and what stops it at the boundary?** The documented skills are all facets of that question — define autonomy levels, classify actions by risk, right-size human intervention, enforce least privilege, and require explicit authorization for irreversible changes (S2).

The answer the exam rewards is **risk-based autonomy**: the amount of human involvement matches the blast radius of the action, not a blanket policy. You met this idea as anti-pattern #7 in Module 1; Domain 6 turns it into a full governance model. Two answer shapes are almost always wrong (S2):

- **"Require human approval for everything."** Slows delivery without reducing material risk — the over-cautious trap.
- **"Let the agent do everything."** Ignores security, compliance, and irreversibility — the over-trusting trap.

The right answer sits between them: scoped permissions, inspectable artifacts, required checks, and human review **only where it materially reduces risk**.

---

2. The autonomy ladder (L0–L5)

"Define autonomy levels" is the first Domain 6 skill bullet (S2). Official docs require you to define levels; they do not hand you a numbered scale, so this course uses a working ladder that maps each level to documented GitHub controls (S2, S11). Learn it as a placement tool: every agent task lands on exactly one rung.

Level	Name	Agent can do	Human role
L0	Suggest	explain, draft, summarize — no writes	decide everything
L1	Plan	produce a plan and artifacts, propose changes	approve before any action
L2	Branch work	create a branch and PR, run tests	review the PR
L3	Low-risk autonomous	merge low-risk changes after required checks pass	monitor exceptions
L4	Gated production	deploy, but only through approved environments	approve deploys and risky operations
L5	Broad autonomous	unscoped write and deploy	**unsafe for most teams — avoid unless heavily controlled**

Read the ladder in both directions. Going up, each level trades review effort for speed and takes on more blast radius. Going down, each level is the fallback when an action fails a risk test at the level above. L5 is on the ladder so you can recognize it in a scenario and reject it: broad autonomous write across repos and environments is the "blanket permissions" risk GitHub's own risk-and-mitigation guidance tells you to mitigate with scoping and gates (S12). An answer that lands a normal team at L5 is a distractor.

The exam phrase to keep verbatim: **assign autonomy levels to maximize delivery speed while remaining compliant** (S2). Both halves count. L1-everything fails the speed half; L5 fails the compliance half.

---

3. Classify actions by operational, security, and compliance risk

Before you can place an action on the ladder, you classify it. Domain 6 names three risk classes (S2):

Risk class	The question	Examples
Operational	Can this break the build, the deploy, or the team's flow?	merging to `main`, editing CI workflows, deleting branches
Security	Can this expose secrets, widen access, or introduce vulnerable code?	touching auth code, editing `.github/workflows/`, adding a dependency, calling a new MCP server
Compliance	Can this violate policy, regulation, or Responsible AI commitments?	handling customer data, changing license files, shipping unreviewed generated code to a regulated surface

Two modifiers cut across all three classes and dominate exam scenarios:

- **Reversibility.** A PR on a branch is cheap to revert; a production deploy, a deleted branch, or a rotated secret is not. Irreversible actions always rank higher (S2, S12).
- **Blast radius.** A docs typo touches one file; a workflow edit touches every future run. GitHub's cloud-agent guidance frames the core risks exactly this way — an agent with too much access or too little supervision can take actions whose consequences outlive the run (S12).

The classification drives the level: low risk and reversible → L2–L3; high risk or irreversible → L4 with explicit authorization; policy-violating → blocked outright (section 5). Write the classification down — a risk table in the repo is an inspectable artifact and the basis of your **risk-based review policy** (S11). You build exactly this table in Lab 6.

---

4. Right-size human intervention

Domain 6 measures whether you can **identify actions requiring human judgment** and configure intervention without slowing delivery (S2). Human judgment is required where a machine check cannot decide the question:

- **Irreversible or hard-to-reverse changes** — deploys, data migrations, secret rotation (S2, S12).
- **Security-sensitive surfaces** — auth code, workflow definitions, dependency and MCP allow-list changes, anything CODEOWNERS should own (S11).
- **Compliance- or policy-ambiguous work** — actions whose acceptability depends on regulation or Responsible AI policy, not on tests passing (S2).
- **Judgment calls the contract didn't anticipate** — scope expansion, contradictory requirements, an agent proposing to change its own guardrails.

Everything outside those categories should flow on automated checks. The placement mechanisms are the ones you already own: **CODEOWNERS** puts a required human on sensitive paths; **environment approvals** put a required human in front of sensitive operations; **branch protection with required checks** lets everything else merge on green (S11). Human-in-the-loop is a placement decision, not a volume decision — Domain 6 is where that Module 1 line becomes the whole exam domain.

---

5. Block what policy prohibits

Some actions are not "high risk, needs approval" — they are **prohibited**, and the skill is to *block* them, not to review them (S2). Blocking means the control fails closed before the action lands:

- **Branch protection** blocks direct pushes and force pushes to `main`; the agent physically cannot skip the PR (S11).
- **Required status checks** block merge until scanning and tests pass (S11).
- **Push protection / secret scanning** blocks commits containing credentials (S11).
- **Actions** `permissions:` blocks a workflow from writing what it was never granted — a job with `contents: read` cannot push, whatever the agent intends (S11).
- **MCP allow lists** block tool calls to servers you never approved (S12).
- **Environment protection rules** block deploy jobs from running without the required reviewer (S11).

Map the policy to the control: security policy violations → scanning, push protection, scoped permissions; compliance violations → branch protection, CODEOWNERS, audit logs proving review happened; Responsible AI policy violations → the agent's operating rules plus a human gate on the surfaces where those policies bite (S2, S12). The distinguishing feature of a block is that no human is asked — the action simply cannot execute. If a scenario says "the agent must never X," the answer is a fail-closed control, not a review step.

---

6. Least privilege and explicit authorization

Two Domain 6 bullets form a pair (S2):

Scope permissions and execution contexts to least privilege. The agent gets the smallest grant that lets the task succeed: read-only tokens unless the task writes; writes confined to a feature branch, never a protected branch; workflow jobs declaring minimal `permissions:` blocks; MCP access limited to allow-listed servers; secrets exposed only to the environments that need them (S11, S12). GitHub's risk guidance is blunt about why — the primary mitigation for an agent misbehaving is that it *couldn't do much damage anyway* (S12).

```
permissions:
  contents: read        # default deny; grant write per job, only where needed
```

Require explicit authorization for irreversible or sensitive changes. Least privilege bounds what the agent *can* do; explicit authorization gates the residue that must remain possible but dangerous. The GitHub-native form is an **environment protection rule with required reviewers**: the deploy job names `environment: production`, and it pauses until a designated human approves — a recorded, attributable decision, not a rubber stamp (S11). Signed commits and audit logs make the authorization trail verifiable after the fact: who approved, what ran, when (S11).

The pair is the accountability half of the domain's title: least privilege limits the damage, explicit authorization attributes the decision, and audit artifacts prove both.

---

7. The guardrail catalogue

Domain 6 scenarios are usually "which control?" questions. These thirteen GitHub-native guardrails cover the documented mitigation surface (S11, S12). Know what each one controls and where it lives:

#	Guardrail	What it controls
1	**CODEOWNERS**	required human review on sensitive paths
2	**Branch protection**	no direct/force pushes; PR + review + checks before merge
3	**Environment approvals**	human gate on deploys and sensitive operations
4	**Actions `permissions:`**	least-privilege token scopes per workflow/job
5	**MCP allow lists**	which tool servers the agent may call
6	**Required status checks**	merge blocked until tests/scans pass
7	**Secret scanning + push protection**	credentials never land in the repo
8	**Code scanning**	vulnerable patterns surfaced before merge
9	**Dependency review**	risky dependency changes flagged on the PR
10	**Signed commits / verified identities**	provenance of every change
11	**Audit logs**	who did what, when — the accountability record
12	**PR templates**	plan/risk/rollback structure forced on every change
13	**Risk-based review policy**	the written rules mapping risk class → autonomy level → required controls

Selection heuristic: match the guardrail to the risk class. *Path-based* risk → CODEOWNERS + branch protection. *Operation-based* risk → environment approvals. *Capability-based* risk → `permissions:` + MCP allow lists. *Content-based* risk → scanning and dependency review. *Accountability* questions → audit logs, signed commits, PR templates. #13 is the meta-guardrail: the policy document that says which of the other twelve applies where — without it, the controls are folklore.

8. Preserve velocity — remove approvals that don't pay rent

The last skill bullet is the counter-intuitive one: **preserve delivery velocity by minimizing approvals that do not reduce material risk** (S2). Every approval gate has a cost — latency, context-switching, and reviewer fatigue that erodes the attention real risks need. An approval earns its place only if a human catching a problem there is plausible and the problem would be expensive.

Audit your gates with three questions:

1. **Would a machine check catch it?** If tests, scanning, or dependency review already fail the bad case, the human adds latency, not safety (S11).

2. **Is it reversible?** A revertable merge on a branch does not need the same ceremony as a deploy (S12).
3. **Has the gate ever caught anything?** A gate that only ever gets clicked through is training reviewers to click through the ones that matter.

Concretely: docs-only and formatting changes with green checks can auto-merge at L3; full-team sign-off on every agent PR gets narrowed to CODEOWNERS paths; a manual approval that duplicates a required check gets deleted. On the exam, an answer that *adds* an approval to a low-risk, reversible action is wrong for the same reason an answer that *removes* the deploy gate is wrong: both ignore risk (S2).

Worked walkthrough — the risk-based review policy

Lab 6 assembles the domain into four artifacts in `gh600-sandbox`:

1. `.github/agent-risk-policy.md` — the risk-classification table: ten agent actions, each classified operational/security/compliance, rated for reversibility, and assigned an autonomy level L0–L5. This is guardrail #13 in writing.
2. `.github/CODEOWNERS` **additions** — required review extended to `/infra/` and `/security/`, so path-based risk gets a named human (S11).
3. **Branch-protection design notes** — PR before merge, required checks, CODEOWNERS review, no force pushes: the L2/L3 boundary as configuration (S11).
4. `.github/workflows/agent-deploy-gate.yml` — a build job with `permissions: contents: read` and a deploy job gated on a protected `production` environment: least privilege plus explicit authorization as running code (S11).

Together they demonstrate every Domain 6 bullet: levels defined, actions classified, humans placed by risk, violations blocked, privileges minimized, irreversible changes gated, and low-risk work flowing untouched.

Recap

- Domain 6 is **risk-based autonomy**: never "approve everything," never "agent does everything."
- Place every task on the **L0–L5 autonomy ladder**; L5 is on the ladder so you can reject it.
- Classify actions by **operational, security, and compliance risk**, weighted by reversibility and blast radius.
- **Right-size human intervention**: human judgment on irreversible, sensitive, and policy-ambiguous actions; automated checks everywhere else.
- **Block** policy violations with fail-closed controls; don't review them.
- **Least privilege** bounds the damage; **explicit authorization** gates and attributes the dangerous residue; **audit artifacts** prove both.
- Know the **thirteen-guardrail catalogue** and match guardrail to risk class.

- **Remove approvals that don't reduce material risk** — velocity is a Domain 6 skill, not a compromise.

Sources

- **S2** — GH-600 study guide (skills measured, Domain 6 bullets, weights) — https://learn.microsoft.com/en-us/credentials/certifications/resources/study-guides/gh-600
- **S11** — Build cloud-agent guardrails (GitHub Docs) — https://docs.github.com/en/copilot/tutorials/cloud-agent/build-guardrails
- **S12** — Cloud-agent risks and mitigations (GitHub Docs) — https://docs.github.com/en/copilot/concepts/agents/cloud-agent/risks-and-mitigations

Where this appears on the exam

Domain 6 is 10–15% of GH-600, and it hybridizes with every other domain — expect guardrail options inside tool-use, orchestration, and evaluation scenarios. Heuristics:

- If a choice **requires human approval for everything**, it kills velocity without reducing material risk — wrong (S2).
- If a choice **lets the agent do everything**, it ignores security and compliance — wrong (S2).
- If an action is **irreversible or production-facing**, the right answer includes an **environment approval or explicit authorization** (S11).
- If an action is **prohibited by policy**, the right answer **blocks** it with a fail-closed control, not a review step (S11, S12).
- The right answer usually combines **scoped permissions + branch-based work + required checks + CODEOWNERS on sensitive paths + environment gates on deploys + audit artifacts** — risk-based autonomy wins (S2, S11, S12).

SPARQIO

GH-600 MASTERCLASS · STUDY NOTES

MODULE 7

Capstone & Exam Readiness

Design, operate, and govern AI coding agents in the GitHub SDLC

Module 7 — Capstone & Exam Readiness

GH-600 Agentic AI Developer Masterclass · Study Guide **Domain focus:** All six domains — capstone integration + exam execution **Est. learner time:** 3–4 hours (guide + capstone lab + timed mock + gap analysis)

> Independent study material. Not affiliated with, endorsed by, or sponsored by GitHub or Microsoft. Contains no exam questions or dumps. Every claim is grounded in official Microsoft Learn and GitHub documentation — see **Sources**.

Learning objectives

By the end of this module you can:

- **Assemble a governed agent SDLC end-to-end** in your sandbox repo, touching all six domains, and grade it against a rubric (S2, S5).
- Recite the **must-know phrase bank** by domain — the exam's controlled vocabulary, automatic under time pressure (S2).
- Recognize **red-flag answer patterns** on sight and prefer the documented **high-yield patterns** (S2).
- Run a **timed mock** under exam-like conditions, with a flagging discipline and domain triage.
- Perform a **gap analysis**: trace every miss to the official source for that skill and close it (S2).
- Make a defensible **booking decision** and execute an **exam-day operating checklist**.

Prerequisites

- Modules 0–6 complete, including every lab. The capstone reuses the artifacts you built in `gh600-sandbox` — Lab 1's governance kit especially.
- `git` and `gh` authenticated (`gh auth status`).
- A 55-minute uninterrupted block for the mock in `60_quiz.md`.

---

1. The capstone: one repo, six domains

Everything this course taught reduces to one sentence you met in Module 0: an agent should produce an inspectable plan before it acts, act only inside approved boundaries, and produce outputs that can be reviewed and evaluated — `plan → act → evaluate`, with GitHub as the **system of record and control**

plane (S2, S4, S5). The capstone is where you stop studying that sentence and ship it.

The spec: in `gh600-sandbox`, assemble a complete governed agent SDLC. One branch, one PR, nine artifacts, every domain represented:

#	Artifact	Domain	Taught in
1	`.github/copilot-instructions.md` — durable operating rules	D1, D3	Modules 1, 3
2	`.github/pull_request_template.md` — structured plan on every PR	D1	Module 1
3	`.github/CODEOWNERS` — review scoped to risk	D1, D6	Modules 1, 6
4	`.github/workflows/agent-plan-gate.yml` — plan → approval → execute	D1	Module 1
5	`docs/agent-tooling/mcp-configuration.json` — MCP allow-list policy	D2	Module 2
6	`AGENTS.md` memory policy — expiration, pruning, reset rules	D3	Module 3
7	`.evals/agent-output-checklist.md` — evaluation signals checklist	D4	Module 4
8	`.github/workflows/capstone-orchestration.yml` — planner → implementer → reviewer	D5	Module 5
9	`GOVERNANCE.md` — risk/autonomy table (L0–L5)	D6	Module 6

Full build steps are in `50_lab.md`. The point is not the file count; it is that a reviewer opening your capstone PR can answer, from artifacts alone: what was the agent allowed to do, what did it plan, what gated the plan, what tools could it call, what does it remember, how is output judged, how do multiple agents avoid collisions, and who authorizes the irreversible actions. That is the entire skills-measured surface in one pull request (S2).

Grading rubric

Points track the official domain weights (S2). Self-grade honestly in the lab; the worksheet records the result.

Domain	Criteria	Points
D1 — Architecture & SDLC	Instructions file states a task contract (inputs / outputs / success criteria); PR template forces Plan / Changes / Tests / Risks / Rollback / Human-review; plan-gate run shows `execute` **waiting for approval** (S5, S10)	20
D2 — Tools & environments	Allow-list policy names only required servers/tools with per-tool permissions; execution context documented (repo scope, branch scope, CI, secrets); workflows carry least-privilege `permissions:` blocks (S6, S7)	25
D3 — Memory & state	Memory policy names surfaces (short-term / long-term / external), sets expiration, pruning, and reset rules, and requires decisions captured as durable artifacts (S9)	10
D4 — Evaluation & tuning	Checklist covers functional, quality, security, process, reliability, and human-review signals; root-cause classification (reasoning error vs tool misuse vs context/environment issue) documented (S2, S12)	15
D5 — Multi-agent	Orchestration workflow runs planner → implementer → reviewer with per-stage artifacts and handoff notes; isolation and arbiter rules written down (S8)	15
D6 — Guardrails	Risk/autonomy table maps L0–L5 to concrete actions and gates; irreversible actions require explicit authorization; no approval exists that fails the "does it reduce material risk?" test (S11, S12)	15
Total		**100**

Pass bar: 85. Below that, the missing points tell you exactly which module to reopen — that is the rubric doing gap analysis for you.

2. The must-know phrase bank

The exam has a controlled vocabulary. Distractors are written in loose synonyms of it; correct answers are written in it. Drill these until they are automatic (S2). All of them are on the night-before flashcard deck (`70_flashcards.md`).

Cross-domain spine: `plan → act → evaluate` · GitHub as **system of record and control plane** · inspectable artifacts · human-in-the-loop (S4, S5).

Domain 1: task contract (inputs / outputs / success criteria) · structured plan · validate and gate · plan job → approval gate → execution job · the seven anti-patterns and their fixes (S5, S10).

Domain 2: smallest sufficient tool set · tool permissions · MCP server · MCP allow list · MCP registry · remote MCP server · execution context · least privilege · branch-based scope · retries, rollback, escalation path, traceability (S6, S7, S8).

Domain 3: short-term / long-term / external memory · durable memory · memory surfaces (`copilot-instructions.md`, `AGENTS.md`, issues, PRs, artifacts) · expiration, pruning, and reset rules · context drift · stale context (S9).

Domain 4: evaluation signals (functional / quality / security / process / reliability / human-review) · logs, plans, traces, outputs, artifacts · root-cause classification: **reasoning error vs tool misuse vs context/environment issue** · evidence-based tuning (S2, S10, S12).

Domain 5: planner → implementer → reviewer · parallel specialists · critic · arbiter / merge coordinator · multi-agent isolation · handoffs and decision records · failed, partial, or stalled executions (S8).

Domain 6: autonomy levels (L0–L5) · risk-based autonomy · operational, security, and compliance risk · explicit authorization for irreversible or sensitive changes · preserve velocity by cutting approvals that do not reduce material risk (S11, S12).

If a phrase in this bank makes you pause, that pause is a gap. Log it in the worksheet and re-read the module section that owns it before you book.

---

3. Red flags to avoid, high-yield patterns to prefer

The exam is scenario-driven: "what is the safest / best next action?" Under time pressure you will not re-derive Domain 1 from first principles — you will pattern-match. So train the patterns deliberately.

Red-flag answer patterns — eliminate on sight

An answer choice is almost certainly wrong if it (S2):

1. **Grants broad, unrestricted write access.** Blast radius is the tell. Least privilege is never the wrong direction (S7).
2. **Allows direct pushes to protected branches.** The PR is the control surface; bypassing it removes review, checks, and audit at once (S11).
3. **Skips plan review for high-risk work.** No write action until the plan is inspected — the Domain 1 spine (S5).
4. **Hides artifacts** — plans in transient chat, logging disabled, output nobody can open later. If a reviewer cannot open it, it does not exist (S5).
5. **Tunes the prompt without evidence.** "Reword the instructions" before reading logs, plans, traces, and artifacts skips root-cause classification — the exact reflex Domain 4 punishes (S12).
6. **Approves everything regardless of risk.** The over-cautious trap. Approving a typo fix is anti-pattern #7, not a safeguard (S2).
7. **Removes auditability** to reduce noise, cost, or friction.
8. **Treats stale memory as truth** instead of refreshing sources and pruning (S9).
9. **Lets multiple agents edit the same scope** with no isolation or coordinator (S8).
10. **Bypasses GitHub controls** — checks, PRs, branch protection, environment gates. If the answer moves governance out of GitHub, it is wrong (S2, S11).

Teaching commentary on the two that catch strong candidates: #5 and #6 are failures of *good* instincts. #5 punishes the pragmatic engineer who fixes prompts fast; the exam wants evidence first, tuning second. #6 punishes the cautious engineer who reaches for maximum safety; the exam wants risk-based placement,

not volume. If you feel virtuous picking an answer, check it against these two.

High-yield answer patterns — prefer on sight

The correct choice usually (S2):

- defines explicit **inputs, outputs, and success criteria**;
- produces a **plan artifact before action** and gates execution on it;
- keeps **GitHub as the audit and control plane**;
- scopes **tools and permissions** to least privilege, with **MCP allow lists**;
- uses **branch protection and PR workflows** as the safety net;
- records **logs, traces, and artifacts** for every action;
- **classifies root causes before tuning** anything;
- coordinates agents with **isolation and handoffs**, an arbiter for conflicts;
- preserves velocity through **risk-based approvals** — gates only where blast radius is high.

When two options both look safe, pick the one that keeps the agent *moving* under controls rather than the one that stops it. The exam certifies operators, not brake pedals (S1, S2).

---

4. Timed-mock strategy

Two mocks, run in this order:

1. **This module's mock** — `60_quiz.md`: 24 original scenario items, distributed by domain weight, **55 minutes**, closed book. It is your calibrated readiness signal, written to the documented skills measured (S2).
2. **Optional volume practice** — the community repo's 41-question mock under **90-minute conditions** (S3). It is a practice aid, never an authority: if its answer conflicts with an official doc, the doc wins.

The documented exam length is 120 minutes (S1) — both mocks train a slightly faster pace than you will need, deliberately.

Operating discipline for any timed run:

- **One pass, flag, move on.** First pass answers everything you can decide in under ~90 seconds; flag the rest. Do not let one scenario eat five minutes.
- **Eliminate red flags first.** Most items lose two options instantly to the red-flag list. A coin-flip between two survivors is 50%; between four, 25%.
- **Domain triage on the second pass.** Spend remaining time on flagged items in your *strong* domains first — those are recoverable points. A flagged item in your weakest domain is a study signal, not a time sink.
- **Answer everything.** Leave nothing blank.
- **Simulate honestly.** No notes, no pauses, no "that one doesn't count." A soft mock produces a soft booking decision.

5. Gap analysis — review every miss against the official source

A mock score is a number; the gap analysis is the value. For **every** miss (and every lucky guess — treat "flagged and got it right" as a miss):

1. **Log it** in the worksheet: question, domain, what you chose, why.
2. **Name the skill bullet.** Each mock item maps to a documented skills-measured bullet — the answer key gives the mapping (S2).
3. **Re-read the official source for that skill** — the S-ID in the key, not a blog, not the community repo, not this course's summary of it. The official doc is what the exam is written against (S2).
4. **Classify your own root cause** the same way Domain 4 classifies agent failures: *reasoning error* (misread the scenario), *vocabulary gap* (did not know the term), or *knowledge gap* (did not know the mechanism). The fix differs: slow down, drill flashcards, or re-do the module lab respectively.
5. **Retest the domain** with that module's quiz before calling the gap closed.

Two or more misses in one domain means the domain is open — reopen that module, redo its lab, then re-mock. One scattered miss per domain with a passing score means you are polishing, not rebuilding.

---

6. The booking decision

Book when all of these are true — checklist form in `80_worksheet.md`:

- [ ] Capstone rubric score **≥ 85/100**, self-graded against artifacts.
- [ ] Mock score **≥ 20/24 (83%)** under honest timed conditions.
- [ ] No domain with two or more unresolved misses.
- [ ] Phrase bank automatic — the night-before deck (`70_flashcards.md`) runs clean.
- [ ] You have re-read the official study guide's skills-measured list end to end and nothing on it surprises you (S2).
- [ ] Exam logistics verified **on the official pages** — see Section 8.

Scoring 17–19/24: book two to three weeks out and spend them on your gap list. Below 17: do not book yet; the readiness table in `60_quiz.md` maps score bands to actions.

7. Exam-day operating checklist

Treat exam day like a production deploy — boring, rehearsed, no novelty:

- **The night before:** run the 16-card deck once. No new material. New material the night before adds anxiety, not knowledge.
- **Verify logistics:** appointment time and time zone, ID requirements, and — for online proctoring — the system test on the machine you will actually use. Pearson VUE's requirements are the authority, checked from the certification page (S1).

- **Environment:** quiet room, cleared desk, stable network. Arrive or log in early.
- **During the exam:** run the mock discipline at exam scale — one pass, flag, eliminate red flags, second pass by strength, answer everything. Budget against the documented 120 minutes (S1).
- **When stuck:** locate the scenario on the loop. A hidden-intent problem is *plan*; a permissions or destructive-command problem is *act*; a stale-context or wrong-conclusion problem is *evaluate*. Then pick the answer that keeps GitHub as the control plane (S2, S5).

8. Final reminder: verify beta/GA status before booking

At the time this course was written, the certification page still labeled GH-600 as **beta**, and community notes suggested a beta window through May 2026 with general availability from July 2026 (S1, S3). Do not trust that from this course or any other. Before you book, open the official certification page (S1) and the Pearson VUE registration flow and confirm the **current** status, duration, price, and passing score yourself. Certification logistics are exactly the kind of fact that goes stale — and verifying at the source is the same operator instinct this exam certifies.

Recap

- The capstone is the course in one PR: nine artifacts, six domains, graded against a 100-point rubric with an 85 pass bar.
- Drill the **phrase bank** — the exam is written in a controlled vocabulary.
- Eliminate **red flags** on sight; prefer the **high-yield pattern**: contract → plan artifact → gate → least privilege → GitHub as control plane → risk-based approvals.
- Mock under honest timed conditions; **flag and move**; triage by domain.
- Gap analysis: every miss traced to its **official source**, classified, and retested.
- Book at **≥ 85 rubric and ≥ 20/24 mock** — after verifying beta/GA status on the official page (S1).

Sources

- **S1** — GitHub Certified: Agentic AI Developer (certification page) — https://learn.microsoft.com/en-us/credentials/certifications/agentic-ai-developer/
- **S2** — GH-600 study guide (skills measured, domain bullets, weights) — https://learn.microsoft.com/en-us/credentials/certifications/resources/study-guides/gh-600
- **S3** — Community study repo (practice aid only — never authority) — https://github.com/jtur671/gh-600-study-guide
- **S4** — Foundations of Agentic AI (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/foundations-agentic-ai/
- **S5** — Design agent architecture & SDLC integration (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/design-agent-architecture-integration/
- **S6** — Agent tooling, MCP & execution environments (Microsoft Learn module) — https://learn.microsoft.com/en-us/training/modules/agent-tooling-mcp-execution-environments/

- **S7** — Prepare for custom agents (GitHub Docs) — https://docs.github.com/en/copilot/how-tos/administer-copilot/manage-for-organization/prepare-for-custom-agents
- **S8** — Custom agents / sub-agent orchestration (GitHub Docs) — https://docs.github.com/en/copilot/how-tos/copilot-sdk/use-copilot-sdk/custom-agents
- **S9** — Copilot memory (GitHub Docs) — https://docs.github.com/en/copilot/concepts/agents/copilot-memory
- **S10** — Implementation planner (GitHub Docs) — https://docs.github.com/en/copilot/tutorials/customization-library/custom-agents/implementation-planner
- **S11** — Build guardrails (GitHub Docs) — https://docs.github.com/en/copilot/tutorials/cloud-agent/build-guardrails
- **S12** — Cloud-agent risks and mitigations (GitHub Docs) — https://docs.github.com/en/copilot/concepts/agents/cloud-agent/risks-and-mitigations

Where this appears on the exam

Everywhere — this module *is* the exam rehearsal. The capstone exercises every skills-measured bullet as running configuration; the mock samples all six domains at their documented weights (S2). If the capstone rubric and the mock both come back green and the gaps are closed against official sources, you are not hoping you are ready — you have evidence. That is `plan → act → evaluate` applied to your own preparation.
