

AI for Embedded Engineers and Makers

A Practical Guide to LLMs, GitHub, Coding Agents, and Repeatable Engineering Workflows

Michael Abate

First edition · Project reference version 1.0.0 · Owner-private final edition

Learn to turn an AI suggestion into a sourced, reviewed, reproducible engineering result.

This book and its companion course develop one sensor-monitor project from a small source excerpt to a documented release candidate. The required path uses synthetic input, host execution, and firmware cross-compilation. Physical hardware is optional.

The linked HTML and EPUB include companion resource texts. Keep the HTML and its assets folder alongside this PDF. Original lab handouts and executable project files belong to the companion learner bundle. Use the linked source for copying long commands; a visual line wrap is not shell continuation syntax.

Contents

AI for Embedded Engineers and Makers 1

Rights and attribution 9

Before you begin 10

Reading map 13

1. AI at the Embedded Workbench 14

2. How LLMs Work: The Useful Mental Model 24

3. Give AI an Engineering Brief 34

4. Set Up Your AI Workbench 44

5. Git Without the Mystery 57

6. Put Your Project on GitHub 70

7. Experiment, Review, and Recover 84

8. Markdown as an Engineering Notebook 99

9. Build a Useful Project Knowledge Base 111

10. Write AGENTS.md for an Embedded Project 124

11. Turn a Repeated Procedure into a Skill 137

12 - Read Datasheets and Plan Interfaces with AI 150

13 - Build Firmware in Small, Verifiable Steps 161

14 - Debug with Evidence 172

15 - Test, Review, and Measure 182

16. Build Your Reusable Engineering Toolkit	195
17. Capstone: Ship a Documented Sensor Monitor	208
18. Maintain the Workflow and Choose What Comes Next	223
Practical references	233
A. Terminal and operating-system notes	234
B. Git task and recovery reference	236
C. Markdown and YAML reference	238
D. Reusable task-brief patterns	240
E. Project records that survive the conversation	242
F. Project instructions and focused skills	244
G. Versions and checkpoint map	246
H. Troubleshooting by evidence boundary	249
I. Sources, rights, and corrections	251
Glossary	253

Lab navigation

6. Lab LAB01: evaluate a second answer
7. Lab LAB02: repeat and compare
7. Lab LAB03: specify a reporting change
7. Lab LAB04: create a setup record
7. Lab LAB05: save a meaningful change
6. Lab LAB06: synchronize and rebuild independently
8. Lab LAB07: review and recover
LAB08: add a recoverable troubleshooting path
LAB09: repair conflicting context
LAB10: test a fresh session
LAB11: build and evaluate the constraints skill
LAB12 - Repair an interface proposal
LAB13 - Expose freshness without changing policy
LAB14 - Produce a defensible fault report
LAB15 - Detect and explain a wrong comparison
LAB16: resume from repository context
LAB17: complete and defend the capstone
LAB18: complete one transfer task

Figure navigation

D01. The engineer defines acceptance; the model proposes; tools act; compiler, tests, and instruments supply different evidence.

SS01-01. The complete explanation request supplies the function and asks for code-supported statements and assumptions before submission.

SS01-02. The actual response identifies equality at 28 degrees; Celsius remains a caller-contract assumption discussed in the worksheet.

SS01-03. The worked evidence review separates actual-response analysis from original teaching claims and keeps unperformed checks pending.

D02. Training adjusts model parameters; inference uses the resulting model with current context. Supplying a project note is not the same as retraining.

D03. The current request, selected history, relevant references, and actual tool results enter bounded context. The generated answer still needs verification.

SS02-01. Without the teaching note, the response conditionally references prior book code and lacks the BenchNote specification.

SS02-02. With the fictional teaching note supplied, the response prioritizes invalidity and distinguishes unspecified recovery behavior.

SS02-03. The actual context comparison preserves the policy facts across repeated prompts and leaves the missing recovery count unspecified.

D04. A goal, relevant context, constraints, and observable acceptance produce a reviewable task. The actual result is compared with acceptance.

SS03-01. The initial request leaves acceptance unspecified.

SS03-02. The brief fixes scope and acceptance.

SS03-03. The actual bounded plan preserves all input records, synthetic origin and the false validity flag.

SS04-01. Windows Arm64 user installer option documented.

SS04-02. Verify the installed extension and publisher.

SS04-03. Inspect the neutral project composer first.

SS04-04. Actual CP00 path and listing shown.

SS04-05. Read permission choices before submitting work.

SS04-06. Completed CP00 IDE task record shown.

D05. Saving changes the working tree; add selects contents in staging; commit records that selected snapshot. Ordinary and cached diffs compare different states.

SS05-01. Git initializes the local practice repository.

SS05-02. Selected source excludes generated build output.

SS05-03. The diff isolates the reporting-label change.

SS05-04. Staging selects the reviewed README change.

SS05-05. Two commits record distinct reviewed purposes.

SS06-01. The intended destination is private.

SS06-02. Inspect the actual origin and branch.

SS06-03. The initial history is pushed successfully and the branch tracks its upstream.

SS06-04. The hosted repository shows the expected files and latest demonstration commit.

D06. Push sends commits to the remote; fetch retrieves remote knowledge without integration; clone creates a separate checkout for reproduction.

SS06-05. Clone the private repository into a separate checkout.

SS06-06. The separate checkout builds successfully using the recorded setup.

D07. A feature develops from a baseline, is reviewed and merged, and a later fault can be reversed by a new commit while the faulty commit remains in history.

SS07-01. Create the feature branch from the correct clean baseline.

SS07-02. Review the intended edit and the unrelated constructed edit.

SS07-03. The private pull request contains only the reviewed README clarification.

SS07-04. Show both competing text changes in the genuine merge conflict.

SS07-05. Resolve the conflict, verify the intended content, and confirm a clean state.

SS07-06. Revert the seeded bad commit and verify recovered behavior.

SS08-01. Record the evidence table for the lesson workflow.

SS08-02. Preview the rendered command, hardware reference, and evidence table.

SS08-03. Follow the README link to the toolchain setup document.

SS08-04. Inspect the finished README for setup, build, expected output, and recovery guidance.

- D08. Requirements, sourced facts, decisions and actual evidence support different parts of a reviewable engineering result.
- SS09-02. Record sourced board and memory facts while leaving physical revision unobserved.
- SS09-03. Record the decision, plan, actual replay result, and handoff together.
- SS09-04. Compare the current hardware and handoff documents as separate context sources.
- SS09-01. Inspect the populated document tree and its linked project context.
- D09. Task instructions and project references guide work; runtime permissions independently constrain available tools and access.
- SS10-02. Inspect the saved guidance and exact AGENTS.md at the Explorer root.
- SS10-03. Begin a fresh task using the repository instructions and exact source context.
- SS10-04. Inspect the recorded command result and changed README inventory after execution.
- SS10-05. Compare the preserved seeded guidance with the corrected replay command.
- SS10-01. Preview the bounded guidance, purpose, links, and commands.
- D10. Skill metadata supports selection; the procedure uses focused references and optional scripts to produce reviewable output.
- SS11-01. Inspect the created skill directory and required SKILL.md.
- SS11-02. Read the completed skill frontmatter and workflow with its focused reference link.
- SS11-03. Inspect sourced constraints after a fresh explicit skill request.
- SS11-04. Inspect the recorded unrelated task after a fresh session with no tool calls.
- SS11-05. Inspect a fresh task that reports missing source material instead of substituting a board identity.
- D11. The optional QT plan maps documented signals; physical assembly, adapter integration, and bench verification remain separate.
- SS12-01. Inspect the exact manufacturer source for product 5477 and its documented memory.
- SS12-02. Manual instructor review compares extracted constraints with source sections, units, and unknowns.
- SS12-03. Variant and units correction records four constructed-claim fixes against the reviewed source.
- SS12-04. Review the bounded interface plan with verified connections, initialization, and open questions.
- SS13-01. Actual CP05 baseline: normal and stale synthetic replays and the firmware build complete with exit 0.
- SS13-02. Actual CP05 formatter diff adds a Boolean current field while retaining the existing state and error fields.
- SS13-03. Actual changed CP05 firmware build, its ESP32-S3 target/options and build directory.
- SS13-04. Actual changed normal replay: eight synthetic sample records, including recovery and the nonfinite fault.
- SS13-05. Actual stale replay: expiry and missing input produce FAULT with current false, followed by recovery.
- SS14-01. Actual original CP06 fixture: the 28C token on line 3 produces invalid sample number and process exit 2.
- SS14-02. Actual read-only AI response proposes a copy-only numeric-token test and explicitly labels the result as predicted.
- SS14-03. The corrected copy exits 0 and reaches NORMAL; the unchanged original still exits 2. The diff shows the single numeric-token change.
- SS14-04. Completed instructor investigation records the supported cause, paired evidence, preserved sources and limits of the synthetic test.
- D12. Shared policy supports host checks and synthetic firmware compilation. A physical adapter needs separate integration and bench evidence.
- SS15-01. Independent CP07 assertions test startup, recovery and inclusive stale equality.

SS15-02. Actual guest CP07 host tests pass all 36 assertions with zero failures and exit 0.

SS15-03. The deliberately premature stale boundary compiles and causes two meaningful assertion failures, with exit 1.

SS15-04. Actual boundary results preserve the passing baseline, two-failure mutation, restored pass and successful build. Injected timestamps are explicitly distinguished from measured device latency.

SS15-05. The evidence report separates three supported software claims from a constructed overclaim and three explicitly unperformed physical procedures.

SS16-01. The actual firmware-review skill states its trigger, bounded procedure and evidence requirements.

SS16-02. Actual read-only review identifies disabled stale expiry, gives trigger and consequence, and distinguishes source predictions from execution.

SS16-04. The fresh repository-guided task reports its bounded negative-infinity change and actual validation. Independent verification also passed 40 assertions.

SS16-03. The instructor handoff preserves case results and receipt locations, identifies limits and gives the next bounded fresh-session task.

SS17-01. The final CP09 reference defines inclusive alert boundaries, stale-input faults, two-sample recovery, bounded commands, and reset behavior.

SS17-04. The synthetic command replay changes threshold and state, preserves rejected settings, and resets to missing input.

SS17-03. The corrected candidate returns NORMAL after the threshold-setting regression and passes all 140 synthetic host assertions.

SS17-05. The private candidate records its local version tag, source commit, release notes and validation limits.

SS17-02. The bounded policy diff adds classification after recovery and after a valid threshold change.

SS18-01. The effort record separates unmeasured active work from a measured execution window and its display holds.

SS18-02. The bounded README clarification distinguishes modeled FAULT from malformed-fixture rejection.

SS18-03. Three maintenance items connect recorded identities to review triggers and relevant checks.

Companion resource navigation

Each resource link opens the corresponding text in the accompanying HTML. Resources are also included within the EPUB. These additional reference texts are outside the 45,000-55,000-word manuscript target.

[book/chapters/ch01/resources/report-state.cpp](#)

[book/chapters/ch02/resources/benchnote-note.md](#)

[book/chapters/ch03/resources/reporting-input.jsonl](#)

[book/chapters/ch07/resources/review-candidate.md](#)

[examples/sensor-monitor/docs/setup.md](#)

[examples/sensor-monitor/dependencies.json](#)

[course/resources/templates/task-brief.md](#)

[examples/sensor-monitor/docs/requirements.md](#)

[course/resources/templates/README.md](#)

[examples/sensor-monitor/AGENTS.md](#)

[course/resources/templates/AGENTS.md](#)
[course/resources/skills/datasheet-to-constraints/SKILL.md](#)
[course/resources/skills/firmware-review/SKILL.md](#)
[examples/sensor-monitor/docs/checkpoints.md](#)
[examples/sensor-monitor/docs/formats.md](#)
[examples/sensor-monitor/docs/hardware.md](#)
[course/resources/templates/project-brief.md](#)
[course/resources/templates/hardware-reference.md](#)
[course/resources/templates/decision.md](#)
[course/resources/templates/test-evidence.md](#)
[course/resources/templates/handoff.md](#)
[course/resources/templates/erratum.md](#)
[course/resources/skills/datasheet-to-constraints/references/extraction-rules.md](#)

Rights and attribution

Copyright © 2026 Michael Abate.

This owner-private final edition contains original instructional prose, examples, diagrams, exercises, and assessments. Product and company names identify the tools discussed. They do not imply endorsement, sponsorship, a publisher affiliation, or ownership of third-party trademarks.

Third-party documentation is linked and attributed near the claims it supports. Its copyright and licenses remain with the respective owners. Do not infer permission to redistribute a complete manufacturer's manual from a link in this book. Dependency and tool licenses travel with their original distributions; retain those notices when using or sharing permitted packages.

The learner project and original resource files are included for the described exercises. Public distribution terms for the book, course, and original code must follow the rights statement supplied with any authorized release; this owner-private edition does not grant a public open-source license. Keep instructor solutions separate when preparing learner-only packages.

Generated narration and captured application screens have their own provenance and rights records in production. A script is not a final transcript, and a planned figure identifier is not an image license or evidence of capture. The released edition must include only assets that completed its rights and privacy review.

No physical validation is implied by a screenshot of a compiler or host test. The scope and limitations of each exercise are part of its teaching result. The optional physical extension is described separately from the required software workflow.

Before you begin

The promise of this book is practical: you will learn to use an assistant while retaining the ability to explain, review, and reproduce the result. You will not be asked to trust a program because its author was an AI, or reject useful assistance because it can be wrong. You will learn to connect a claim to the code, source document, or observation that can support it.

The intended reader has basic firmware experience. You should recognize a function, an input, a conditional, and an output. You need not already understand Git, Markdown, agent instructions, or skills. Those are introduced in sequence. The [glossary](#) is available whenever a new term interrupts the flow, and the [terminal reference](#) explains the small command vocabulary used here.

The book is about AI-assisted development. The reference firmware does not need to run a language model or contact a cloud service. Its core behavior is ordinary C++ policy, parsing, reporting, and timing logic. That makes the engineering decisions visible while we learn how to work with a model and its tools.

What you need

Use a computer that can run the selected editor, Git, the supported native C++ compiler, and Arduino CLI with the recorded board platform. The demonstrated learner environment is Windows 11 ARM under a standard user account. The [supported-version reference](#) distinguishes that path from the separately checked macOS host route and unverified Linux guidance. A similarly named application or a different architecture is not automatically the tested combination.

The selected AI path is browser chat followed by Codex in VS Code using the available subscription sign-in route. No separately billed API client or project API key is required. Account eligibility, limits, and interface controls can change; check the current official guidance linked in the setup chapter and your actual account. Saved original reference responses let you practice analysis when a live interaction is unavailable. Label those exercises honestly rather than inventing a new model run.

Use a GitHub account you control for the private practice repository. Public publication is unnecessary. The book's demonstration accounts are separate from the author's working accounts. You should similarly understand which project contents you share and keep credentials and unrelated material out of learning artifacts. Private sign-in intervals are intentionally omitted from recordings.

No board, sensor, cable, bench instrument, or hardware purchase is required for the core chapters, course, or capstone. The documented example target is the ESP32-S3 Feather variant identified in the project hardware reference, with a BME280 as an optional future acquisition target. The required firmware acquires synthetic values. Cross-compilation and host tests do not turn those values into physical measurements.

How to use the chapters and lessons

Read each chapter's outcome and starting checkpoint first. Work through the example, then attempt the lab before consulting the separate instructor key. Finish the checklist, three explanatory review questions, and transfer task. A chapter lab produces an artifact such as a brief, a reviewed commit, a constraints table, or a test record. Keep those artifacts together in your learner project or notebook.

The course has eight modules with five lessons each. A lesson focuses on one action and an observable check. Watch or read the demonstration, pause to perform the exercise, and compare your result with its criteria. The lesson text remains useful when you cannot play media. Narration and captions support the same content; neither replaces the actual commands, files, and evidence you produce.

Each module ends with a challenge and a five-question self-check. Four correct answers is the recommended quiz threshold, with unrestricted retry and immediate explanations. This is a learning aid, not a professional credential. A progress acknowledgment records that you reviewed a lesson; the practical artifact determines whether you performed its engineering task.

In the standalone website, progress belongs to your browser. Use its export function before clearing browser data or moving to another device, and import your saved progress when needed. Reset deliberately when you want a fresh self-study record. The LMS package stores its attempt through the LMS instead; do not assume a browser-local progress file and an LMS attempt are the same record. Neither kind of progress is evidence that a board was physically tested.

Use the chapter resources at their named stage. CP00 is an inspection packet. CP01 is a fixed synthetic host/firmware baseline with no automated assertions or threshold policy. Later stages add documents, instructions, skills, normal/fault behavior, tests, and finally the configurable capstone. Extract each checkpoint into a new exercise folder rather than overwrite useful work. ZIPs provide content; the separate generic history bundle provides the graph needed for branch, conflict, and revert exercises.

Some introductory examples are deliberately simpler than the final project. Chapter 1's original string-return function and Chapter 2's fictional BenchNote note use OK/WARN/FAULT. CP00 and the final reference project have their own defined state names. Chapter 3 includes an original validity fixture distinct from CP00's log. The text names these boundaries so that a useful simplification does not become an accidental project requirement.

What counts as evidence

Documented means a specific applicable reference supports the claim. **Code-supported** means the supplied source supports the statement within its input assumptions. **Observed or tested** means a named action actually ran and its result was recorded. **Synthetic** means the input was deliberately generated for software work. **Illustrative** means a passage or output was constructed to explain an idea. **Unresolved** means the current material is insufficient to decide the claim.

Expected output is a prediction or acceptance criterion until your command executes. Preserve actual failures and their context. If a video is an edited replay, it should be labeled accordingly; editing for readability must not change the displayed test result or imply an uncut chronology. The [evidence reference](#) provides a small record format that keeps input, action, result, and limitation together.

A compiler checks a program against its build rules and selected target. A host test exercises specified software behavior. A synthetic replay demonstrates that behavior for controlled events. A primary document can establish intended pin or interface constraints. A physical instrument measures an actual assembly. Each is valuable, and each has a scope. The habit of saying exactly what passed is central to the course.

Keep your own pace. Installation, reading, and debugging take as long as they take. Short recordings can omit waiting intervals with an explicit transition; that is not a promise that your machine will finish in the same time. If a check fails, use the troubleshooting index to classify the problem, preserve the evidence, and continue independent work where possible. Your aim is a result you understand well enough to maintain.

Reading map

Part	Chapters	What you will produce
Understand AI	1-4	Evidence annotation, context comparison, engineering brief, verified workbench record
Git and GitHub	5-7	Meaningful commits, private synchronization, reproduction and recovery evidence
Project context and skills	8-11	Useful README, sourced project records, tested instructions, focused skill
Apply the workflow	12-15	Interface constraints, incremental firmware, diagnosis, tests and validation limits
Make it your own	16-18	Reusable toolkit, capstone evidence bundle, maintenance and transfer plan

The full generated table of contents and PDF bookmarks follow the assembled chapter order. The appendices are task references rather than additional prerequisites: consult the [appendix index](#) when a specific interruption arises.

1. AI at the Embedded Workbench

After this chapter, you will be able to choose a useful AI task and identify the evidence needed to accept its answer. You need basic familiarity with a firmware program: a function accepts inputs, performs work, and produces an output. You do not need a board, paid API access, or previous experience with a coding agent. Start with checkpoint CP00 and keep a plain text file named `evidence-annotation.md` beside your copy of its resources.

Imagine that a temperature monitor reports a plausible number but never raises its warning. An assistant suggests changing a comparison, replacing the sensor library, and moving a wire. Those suggestions involve three different kinds of knowledge. The comparison is in your code. The library interface belongs to a particular software version. The wire exists on a physical assembly. A fluent paragraph can mix them so smoothly that the distinctions disappear. Your first engineering task is to put the distinctions back.

This book develops a sensor monitor because it gives us a small, understandable system with real decisions to make. The central skills apply equally to a motor controller, garden logger, instrument, or classroom robot. We will use AI to reduce the effort of understanding and changing the project, while keeping its behavior explainable. The finished result includes code, requirements, tests, a useful history, and instructions that let another session resume the work.

1. Choose assistance with a visible result

AI is useful when you can give it relevant material and assess what comes back. Ask it to explain an unfamiliar function, identify missing cases in a test table, compare a change against a requirement, or turn a rough setup note into a sequence you can actually follow. Each request has an artifact you can examine. An explanation names lines. A test table lists inputs and expected outputs. A review connects a finding to the changed code. A setup guide succeeds when a second checkout can follow it.

Start with work you can already judge. If you know enough C++ to recognize a changed comparison, use the assistant to explain it and draft a boundary test. If you have never read a sensor datasheet, ask the assistant to help locate the relevant section and define the terms. Do not make its first answer the sole basis for a connection you do not understand. There is value in having a patient guide, but the guide's confidence does not increase a component's voltage rating.

Choose the size of the task to fit your review capacity. “Design the complete firmware” creates a large result with many hidden decisions. “Explain how this function decides whether to raise a warning, without changing files” is small enough to check. Once that explanation is sound, a second request can ask for a specific change. This sequence preserves your ability to stop at a clear boundary.

Useful assistance also includes saying what is missing. If a request asks whether a library call is supported but omits the library version, a good response identifies the missing version. Treat an explicit unknown as progress when it prevents an unsupported decision. The next step is now

concrete: inspect the dependency record or official reference, rather than ask for a more confident answer.

2. Separate model, application, and agent

A **model** is the learned system that processes input and generates output. An **application** supplies the interface around it: a chat box, attachments, conversation history, account settings, and available tools. An **agent** is a workflow in which model output can lead to actions, such as reading a file, executing a command, or making an edit. The exact capabilities depend on the application and its permissions. The official Codex IDE documentation describes an editor interface connected to project work; it does not make every chat window an editor agent. [Codex IDE extension](#)

These distinctions resolve a common misunderstanding. A browser answer can contain a shell command without having executed it. An agent can report that it ran a command, but you still need to inspect the tool result and execution context. A compiler can successfully build a different folder from the one you intended. “The AI checked it” is therefore an incomplete record. Name the action, file or folder, command, and result.

The interface can help you see those details, but an attractive summary is not a substitute for them. During later lessons, you will ask the agent to identify the files it inspected and will compare its edit with a Git diff. A diff is a view of changes between versions. You will also run selected checks independently of the agent's final message. That does not require repeating every action; it means checking the evidence that matters to the decision.

We use browser chat for the first conceptual exercises and Codex in a project editor for subsequent work. Other assistants can support the same habits, but their controls, instruction discovery, account limits, and access rules differ. Follow the tested path first. Once you have a representative task and a verification method, you can compare tools without changing the meaning of success.

3. AI at development time and AI on a device

Our monitor uses AI during development. The learner asks questions, edits source, reviews changes, and tests behavior with assistance. The resulting firmware need not contain a language model, contact a cloud service, or perform machine learning. Removing the assistant from the workbench does not remove a threshold comparison from the compiled program.

Running AI on an embedded device is a separate design problem. It introduces questions about model size, memory, timing, energy, input quality, and how inference affects system behavior. Those can be worthwhile projects, but they are not prerequisites for this one. Keeping the two activities distinct lets you use an assistant on an ordinary embedded program without claiming the program itself is an AI product.

The distinction also improves debugging. If a host test fails because a comparison excludes the threshold value, the failure belongs to the software requirement and implementation. You do not need to understand model training to investigate it. If the assistant proposed the

comparison, that explains how the defect entered the workflow; it does not change the technical evidence needed to fix it.

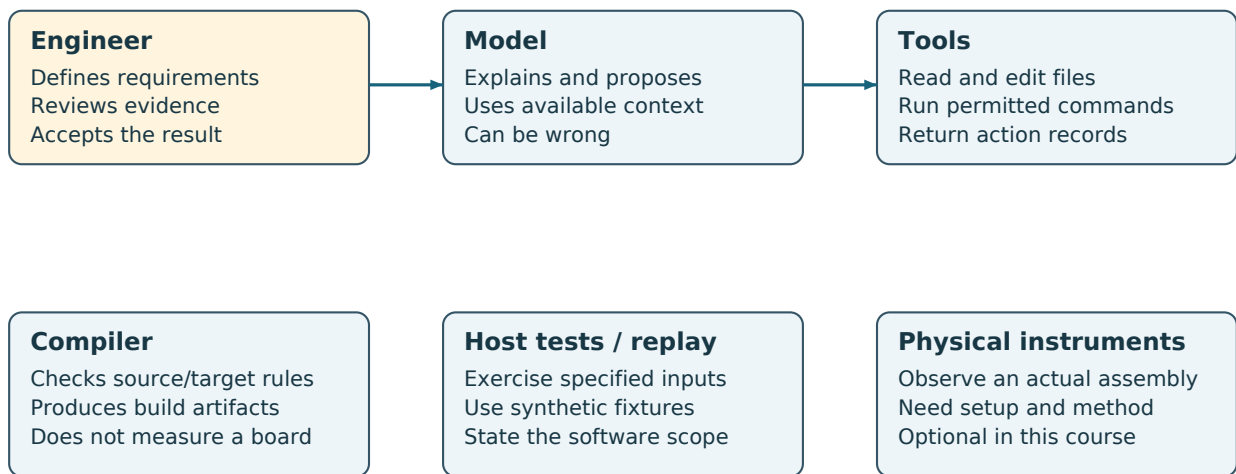
4. Allocate responsibility before accepting a claim

Use three labels in your notebook. **Code-supported** means the supplied source directly supports the statement, subject to its stated inputs and dependencies. **Document-dependent** means the statement requires a particular reference, such as a library API or board schematic. **Measurement-required** means a physical observation is necessary to establish the claim for the actual assembly. Add **unresolved** whenever the available material does not justify a conclusion.

These labels are about the current evidence, not permanent categories for all time. A pin assignment can move from unresolved to document-supported when you inspect the exact board revision. A claim about successful electrical communication remains unmeasured until you run the appropriate physical check. A source document can describe what should happen; your assembly can still be wired incorrectly.

Figure D01 depicts this division of work: the engineer defines and judges the requirement; the model proposes or explains; tools read and modify artifacts; the compiler checks a program against its build rules; tests exercise specified behavior; instruments observe physical signals. The figure's arrows represent exchanges of information, not a transfer of responsibility. A passing check has a scope.

D01 · Distinct responsibilities at the workbench



Evidence returns to the engineer for a bounded decision. No single check establishes every claim.

D01. The engineer defines acceptance; the model proposes; tools act; compiler, tests, and instruments supply different evidence.

For example, compilation can detect many language and interface errors, but it cannot establish that a connector is seated. A synthetic input test can establish that the compiled policy returns the expected state for supplied numbers, but it cannot establish sensor accuracy. Reading a schematic can identify the intended signal path, but it cannot measure a broken trace. State the useful positive result, then state the remaining question.

5. Worked example: annotate a small program

The following is an original teaching excerpt, not a complete sensor driver. Its input is already a number. A copy is supplied as `report-state.cpp`. CP00 uses a separate inclusive-comparison excerpt returning a Boolean; compare the two ideas without treating their function names or return types as identical.

```
const char* report_state(float temperature_c) {
    if (temperature_c >= 28.0f) {
        return "WARN";
    }
    return "OK";
}
```

Ask a browser assistant: “Explain this function in plain language. Separate statements directly supported by the code from assumptions that require another source or a physical test. Do not change it.” Supply the complete excerpt. Keep the response as received. Your wording may differ from the instructor's saved response; the exercise assesses the claims, not whether the model repeats a particular sentence.

Here is an intentionally mixed **illustrative response**, written for this exercise: “The function returns WARN at 28 degrees Celsius or above and OK below that value. It samples a BME280 every second through the board's default I2C pins. The measurement is accurate to the sensor's specification, so the device is safe to deploy.” This is not a recorded model response. It contains useful statements and unsupported additions on purpose.

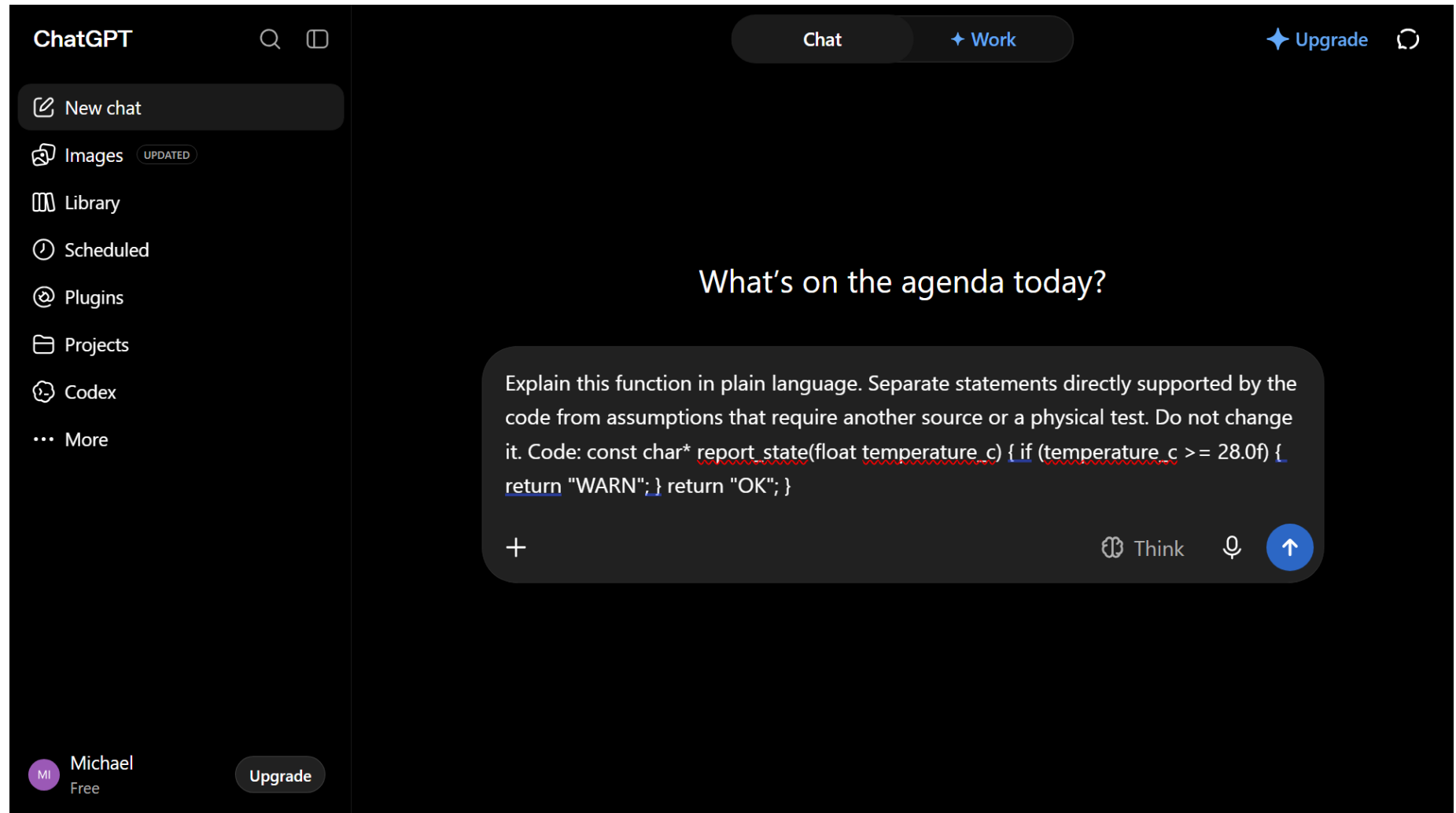
Annotate the first sentence as code-supported for ordinary finite numeric inputs: `>=` includes equality, and the two string literals identify the returned labels. The parameter name expresses an intended unit, but the function cannot prove that its caller actually supplies Celsius. Record that caller contract separately. The function contains neither a sensor read nor a scheduler, so the sampling and I2C claims are unsupported by this excerpt. Exact interface details would require the full source and target documentation.

The accuracy claim needs much more than a part name. You would need a specific documented condition, the actual sensor configuration, and measurements suitable for the intended use. The deployment conclusion does not follow from this function. Mark it unresolved and replace it with a bounded statement: “The comparison implements the stated threshold for the shown inputs; acquisition, invalid values, timing, and physical operation are not established here.”

Your worksheet can use these columns: claim; current label; supporting line or source; next check; result. For the first row, the next check is to evaluate 27.9, 28.0, and 28.1. For the sampling row, inspect the caller and its timing code. For pin mapping, locate the exact board

documentation. For physical accuracy, define a bench comparison if you elect to perform the optional hardware work. Avoid a vague “verify later” that leaves nobody knowing what to do.

SS01-01 shows the submitted request; SS01-02 preserves the actual answer. SS01-03 is the instructor's annotated worksheet. Its constructed teaching claims remain separate from the recorded response; none establishes physical operation.



SS01-01. The complete explanation request supplies the function and asks for code-supported statements and assumptions before submission.

ChatGPT 🔍 📄

New chat

Images UPDATED

Library

Scheduled

Plugins

Projects

Codex

... More

Recents

Explain function behavior

Michael Free **Upgrade**

Explain this function in plain language. Separate statements directly supported by the code from assumptions that require another source or a physical test. Do not change it. Code: `const char* report_state(float temperature_c) { if (temperature_c >= 28.0f) { return "WARN"; } return "OK"; }`

The function takes a temperature in **degrees Celsius** and returns one of two text labels: `"WARN"` or `"OK"`.

Directly supported by the code

- `temperature_c` is a `float`, so the function expects a numeric temperature value.
- If `temperature_c` is **28.0°C or higher**, it returns `"WARN"`.
- If `temperature_c` is **below 28.0°C**, it returns `"OK"`.
- The comparison uses `>=`, so **exactly 28.0°C produces "WARN"**.
- The function does not modify the temperature or perform any other calculations.

In plain language:

"If the temperature is at least 28°C, report a warning; otherwise, report OK."

↓

Assumptions that need another source or physical test

+ Ask anything Think 🗣️ 🎧

SS01-02. The actual response identifies equality at 28 degrees; Celsius remains a caller-contract assumption discussed in the worksheet.

The screenshot shows a Visual Studio Code window with the following elements:

- Address Bar:** 01-evidence.md - capture-packets-m01-wave01 - Visual Studio Code
- Top Bar:** Restricted Mode is intended for safe code browsing. Trust this folder to enable all features. [Manage](#) [Learn More](#)
- Tab Bar:** Welcome x 01-evidence.md x 02-task.md x
- Breadcrumbs:** resources > m01-l01 > 01-evidence.md
- Markdown Preview:** Evidence review — instructor worked example
- Content:**
 - Actual-answer analysis and separate original teaching claims.
 - ACTUAL: 28.0 gives WARN (finite input).**
Support: `>= 28.0f`. Source inspection; no guest test run here.
 - ACTUAL: Celsius needs a caller contract.**
The name suggests units; the supplied excerpt has no caller.
 - ORIGINAL: BME280 sampling, timing, and pin claims.**
No acquisition or timer shown. Next: caller and exact board pinout.
 - ORIGINAL: accurate sensor / ready to deploy.**
No physical result. Next: a defined separate physical comparison.
 - Caller, board, timing, and physical checks remain **PENDING**.
- Bottom Bar:** Restricted Mode, 0 errors, 0 warnings, 0 info, Screen Reader Optimized

SS01-03. The worked evidence review separates actual-response analysis from original teaching claims and keeps unperformed checks pending.

6. Lab LAB01: evaluate a second answer

Start a new section in `evidence-annotation.md`. Consider this original scenario: a reporting routine receives `{valid: false, temperature_c: 31.0}` from a synthetic fixture. A proposed explanation says, “The room is hot, the alarm must turn on, and the sensor cable is disconnected.” Your independent task is to classify those three claims and propose a better response. You may ask an assistant to help, but write your own final judgment.

First identify which part of the input is an observation and what produced it. The fixture is authored software data, not a room measurement. Second decide whether the invalid flag should affect the interpretation of the temperature. You have not yet been given a complete fault policy, so record that requirement as missing. Third ask whether this tuple can uniquely establish cable disconnection. It cannot distinguish that physical explanation from a failed conversion, a deliberate test input, or another source of invalid data.

The expected observation is a worksheet that preserves the invalid flag, labels the input synthetic, and declines to treat a numeric field as a trustworthy reading merely because it is present. A strong revised response says: “This fixture represents an invalid sample. Consult the fault requirement before selecting a state. It does not establish room temperature or a physical cause.” Save that response and the next check you would perform.

7. Failure and recovery: an unsupported pin claim

Suppose the assistant supplies a pin number and cannot identify a reference. Do not turn the number into a wiring instruction by repeating it in a README. Preserve the proposed value as an unverified claim, identify the exact board variant, and inspect its manufacturer pinout and schematic. The Feather family includes variants; the example target in this course is the documented ESP32-S3 Feather. [Adafruit target guide](#)

If the source is unavailable, stop that connection decision and continue work that does not depend on it: annotate the policy, write host tests, or refine the task brief. The recovery is complete when the claim has a traceable source or is explicitly unresolved. Inventing a plausible number to keep moving makes the later work less reliable.

Completion checklist

- ☐ I identified three useful assistance tasks and an observable check for each.
- ☐ I separated model output, tool actions, compilation, synthetic tests, and physical evidence.
- ☐ I annotated the worked response and completed LAB01 independently.
- ☐ I replaced an unsupported conclusion with a bounded statement and next action.

Review questions with answers

1. Why is a correct code explanation insufficient evidence of a working sensor? It addresses the supplied program text. Sensor operation also depends on the acquisition code, configuration, electrical assembly, and actual communication. The explanation is useful within its scope, but those additional claims need their own checks.

2. Is an answer that reports missing information a failure? Not when the information affects the decision. Naming the missing board revision or fault requirement turns uncertainty into a specific research or design task. Guessing would hide that task.

3. Can a synthetic test be valid evidence? Yes. It can establish software behavior for controlled input and is central to this course. It becomes misleading only when described as a physical measurement or used to support claims the test did not exercise.

Transfer exercise

Choose one task from a project you already understand. Write a five-line note naming the task, material you can supply, result you want, evidence you will accept, and an explicit unknown. Prefer a task you can review in one sitting. Keep the note: Chapter 3 will turn it into an engineering brief.

2. How LLMs Work: The Useful Mental Model

This chapter gives you enough of a language-model mental model to prepare better requests and interpret answers without guessing at hidden reasoning. You will compare answers with and without a short reference passage, then record what the comparison establishes. Begin after Chapter 1 with a browser chat or the supplied original sample responses. No machine-learning mathematics or API account is required.

An assistant can explain a familiar programming idiom beautifully, then invent an option for a library it has never inspected. That is less mysterious when you distinguish generating a likely answer from looking up a verified record. The useful question is not whether the system sounds knowledgeable. It is what information was available, how the answer used it, and which claims survive an independent check.

1. Tokens and next-token generation

A **token** is a unit of model input or output. It can be a word, part of a word, punctuation, or another encoded unit, depending on the tokenizer and input. Do not equate one token with one English word. Code, identifiers, numbers, and unfamiliar strings can divide differently. Token counts matter because systems have limits on the material they can process and produce in an interaction. [OpenAI key concepts](#)

A practical simplified model is that a language model repeatedly predicts a next token using the preceding context, then continues from the growing sequence. Learned patterns can support explanations, transformations, and useful problem solving. The generation process does not automatically consult the current datasheet for a part number or execute a program whose output it describes. Those require supplied references or tools.

Think of a generated command as a proposed piece of text until there is evidence of execution. If an answer shows PASS, that string might be an illustrative expected result. When a terminal actually runs a test, the result belongs to a particular executable, input, and environment. Both kinds of text can look identical on a page. Our evidence labels distinguish them.

This mental model is intentionally limited. Modern systems may use several kinds of models, additional processing, retrieval, and tools. We do not need to infer their private implementation to use them well. We need to recognize that the answer depends on information and constraints supplied to a generation process, and that technical correctness must be checked against the relevant requirement and evidence.

2. Embeddings and attention, intuitively

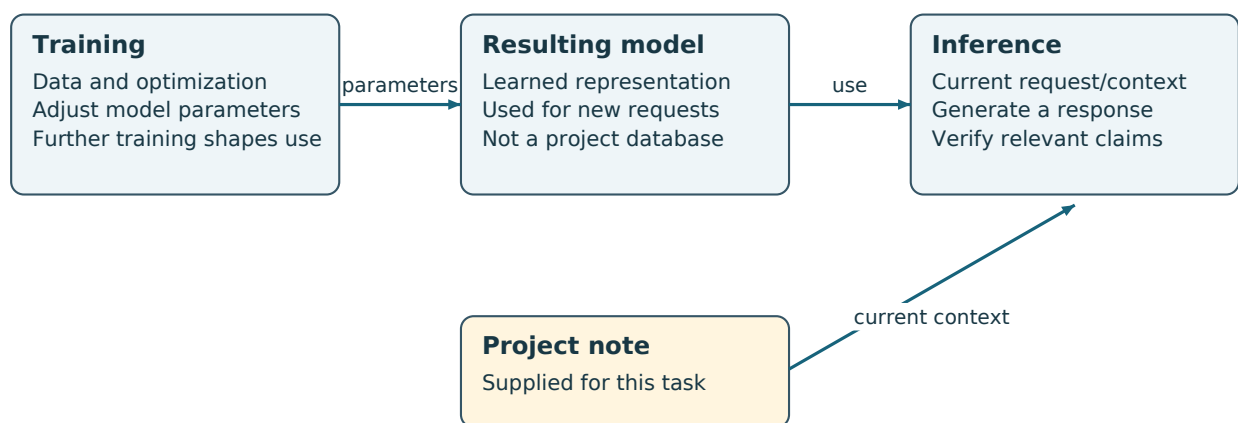
An **embedding** is a numerical representation that allows a model or retrieval system to work with relationships among inputs. In ordinary language, related meanings can have useful relationships in that representation. It is not a built-in certificate that two similarly named components are electrically interchangeable. Similar names are precisely where a maker should be careful: a familiar family name can invite the wrong variant assumption.

Attention is a mechanism that lets a model weight relationships among elements of its input representation. The original Transformer paper introduced an architecture centered on attention mechanisms. This supports a conceptual picture of information from several parts of an input contributing to an output; it does not imply that every relevant sentence is always used correctly. Vaswani and colleagues, *Attention Is All You Need*

Imagine a short project note that states a threshold, unit, and fault behavior. The assistant can connect those details when answering a question about an invalid sample near the boundary. Now imagine burying that note among thousands of lines of unrelated logs and an older contradictory requirement. More material has been supplied, but deciding which statement governs has become harder. Good context is selected and organized, not simply large.

Figure D02 separates learning stages from the current interaction. Figure D03 shows the current request, relevant references, previous messages that are actually retained, and tool results entering a simplified context flow. Neither diagram represents a particular model's internal trace. Their purpose is to show where you can improve the information presented to the system.

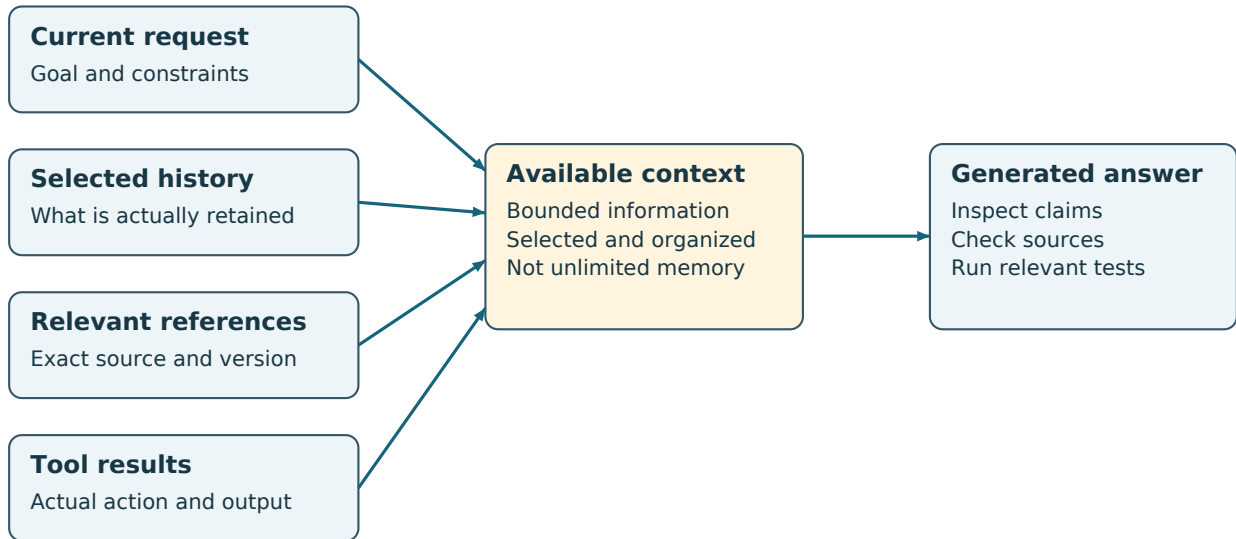
D02 · Training and the current interaction



Conceptual simplification, not a diagram of a particular model's private implementation.

D02. Training adjusts model parameters; inference uses the resulting model with current context. Supplying a project note is not the same as retraining.

D03 · What enters the available context



D03. The current request, selected history, relevant references, and actual tool results enter bounded context. The generated answer still needs verification.

3. Training, post-training, and inference

Training adjusts model parameters using data and an optimization process. **Post-training** is further work that shapes how a trained model behaves, for example in following instructions. **Inference** is using the resulting model to process new input and produce output. These terms explain why putting a technical passage into a conversation can affect the next answer without being the same operation as retraining the model.

For the maker, the practical distinction is immediate. If you provide today's build error, that error can inform the current response. You have not thereby updated every future session's knowledge of your project. If you correct a pin mapping in one conversation, a later session may need the corrected project document again. Store durable facts and decisions in the repository, where both people and future tools can inspect them.

Similarly, a model's learned knowledge is not guaranteed to include the latest release of a library or your private circuit modification. Avoid asking it to infer the installed version from a product's general popularity. Supply the exact dependency record and relevant source. The more consequential the version difference, the more important it is to put that version into the task rather than assume the assistant will ask.

This also changes how you interpret a correction. A model can agree with feedback that is wrong. Agreement is not independent confirmation. If you say that a threshold should include equality, the real question remains what the requirement says and whether the implementation satisfies it. Feedback should point to evidence, not merely insist on a preferred answer.

4. Context windows and missing history

The **context window** is the bounded amount of information available to a model in an interaction. An application may manage long histories by selecting messages or summarizing earlier material. Different products implement this differently. A visible conversation is therefore not a promise that every previous detail remains available in its original form. OpenAI's conversation-state documentation describes context limits and techniques for managing long interactions. [Conversation state](#)

Treat critical constraints as project artifacts, not memories you hope the assistant retains. A short task brief can name the requirement document and test command. A handoff can state the current commit, relevant files, open question, and next action. Later chapters build those artifacts. For now, put the essential excerpt directly into the request and ask the assistant to distinguish supplied facts from assumptions.

An effective context packet has enough detail to decide the task and little unrelated material. Include the failing input, expected output, actual output, affected function, versions, and constraints. Omit a full day's log when three lines reproduce the symptom. Keep the full raw evidence locally, but supply the small relevant portion with its origin and any filtering stated. This makes the response easier for you to audit too.

If a conversation drifts, do not argue indefinitely with accumulated assumptions. Save the useful evidence, prepare a concise fresh brief, and start a new interaction. This is a controlled restart, not a trick for improving luck. You should be able to explain what information changed and why the new request is better defined.

5. Variation, hallucination, and confidence

Generated responses may differ across runs. Wording can vary without changing the technical meaning, and a repeated error can remain stable. Do not score quality by exact textual agreement. Compare claims, cited support, and whether the requested artifact satisfies its acceptance criteria. The OpenAI text-generation guide documents generation as model output controlled by input and model settings; it is not a guarantee of reproducible engineering truth. [Text generation](#)

We use **hallucination** here for a generated claim presented as factual when it is unsupported or false, such as a nonexistent API option or invented source section. The word names an output problem; it does not explain every cause. Your recovery is still practical: isolate the claim, inspect the authoritative material, and correct the resulting artifact.

Confidence language is a poor measurement instrument. "Definitely," a detailed explanation, or an answer repeated by a second assistant can make a claim feel stronger without adding evidence. A second review can be useful because it may identify a missed case or alternative hypothesis. Its value comes from the check it proposes and the evidence it finds, not from counting votes among models.

Ask for a concise rationale, relevant file locations, and verification steps. You do not need a claimed transcript of hidden reasoning. A readable explanation can help you judge a result, but

it is itself another output to evaluate. A test with a deliberately wrong comparison will teach you more about a boundary than a page of confident prose defending it.

6. Worked example: give the answer a source

Use this original chapter resource, [BenchNote technical note](#). It is separate from CP00's shorter inclusive-comparison note. It describes a fictional teaching instrument called the **BenchNote monitor**; it is not a manufacturer specification:

BenchNote accepts a synthetic sample once every 1,000 milliseconds. A sample carries a validity flag and a temperature in degrees Celsius. An invalid sample produces FAULT and its numeric temperature must not select OK or WARN. For this introductory exercise, a valid sample at or above 28.0 degrees produces WARN; a valid sample below 28.0 produces OK. This simplified policy has no hysteresis. It is not the complete capstone requirement.

In a fresh chat, first ask: “How does the BenchNote monitor handle an invalid temperature sample?” Do not supply the note yet. A sound answer may say it lacks the specification. Another may invent a typical behavior. Save the actual answer and label this the **without-reference run**. Do not lead the assistant by saying that you want a particular fault state.

Then provide the complete original note and ask the same question with this addition: “Use only this note for product-specific claims. Identify what remains unspecified.” Save the answer as the **with-reference run**. Now the output can be checked against explicit sentences. It should identify FAULT and explain that the invalid number must not select the ordinary states. It should not assert a physical cause, an I2C address, or a recovery delay absent from the note.

These are **expected response properties**, not a transcript of a recorded chat. If the actual model does something different, preserve it and analyze the difference. A refusal to guess in the first run is a good result. A correct second answer demonstrates that this answer used the supplied material appropriately; it does not establish that every future answer will do so.

Make a table with each product-specific claim as a row. Add columns for answer without reference, answer with reference, supporting sentence, and unresolved detail. The key change should be traceability, not just length. A long answer with an invented restart timer is less useful than two correct sentences that acknowledge the missing timer.

SS02-01 preserves the actual baseline: the new chat conditionally reused earlier code context without a BenchNote rule. SS02-02 shows the supplied-note answer. SS02-03 is the instructor's comparison of that answer and its repeat; both leave recovery unspecified. These observations establish no reliability rate.

The screenshot shows the ChatGPT web interface. On the left sidebar, the 'New chat' button is visible, along with a list of recent chats including 'Invalid Temperature Handling' and 'Explain function behavior'. The main chat area displays a user query in a blue bubble: 'How does the BenchNote monitor handle an invalid temperature sample?'. The AI response explains that invalid temperature samples are not explicitly handled, showing a code snippet for a temperature check function. The response also mentions that values like NaN or out-of-range temperatures could produce 'OK' or 'WARN' results. At the bottom, there is a search bar with the text 'Ask anything' and a 'Think' button.

ChatGPT

Upgrade Share

New chat

Images UPDATED

Library

Scheduled

Plugins

Projects

Codex

More

Recents

Invalid Temperature Handling

Explain function behavior

Michael Free Upgrade

How does the BenchNote monitor handle an invalid temperature sample?

If you mean the temperature-monitoring code you shared earlier, an **invalid temperature sample is not explicitly handled**.

The function only does this:

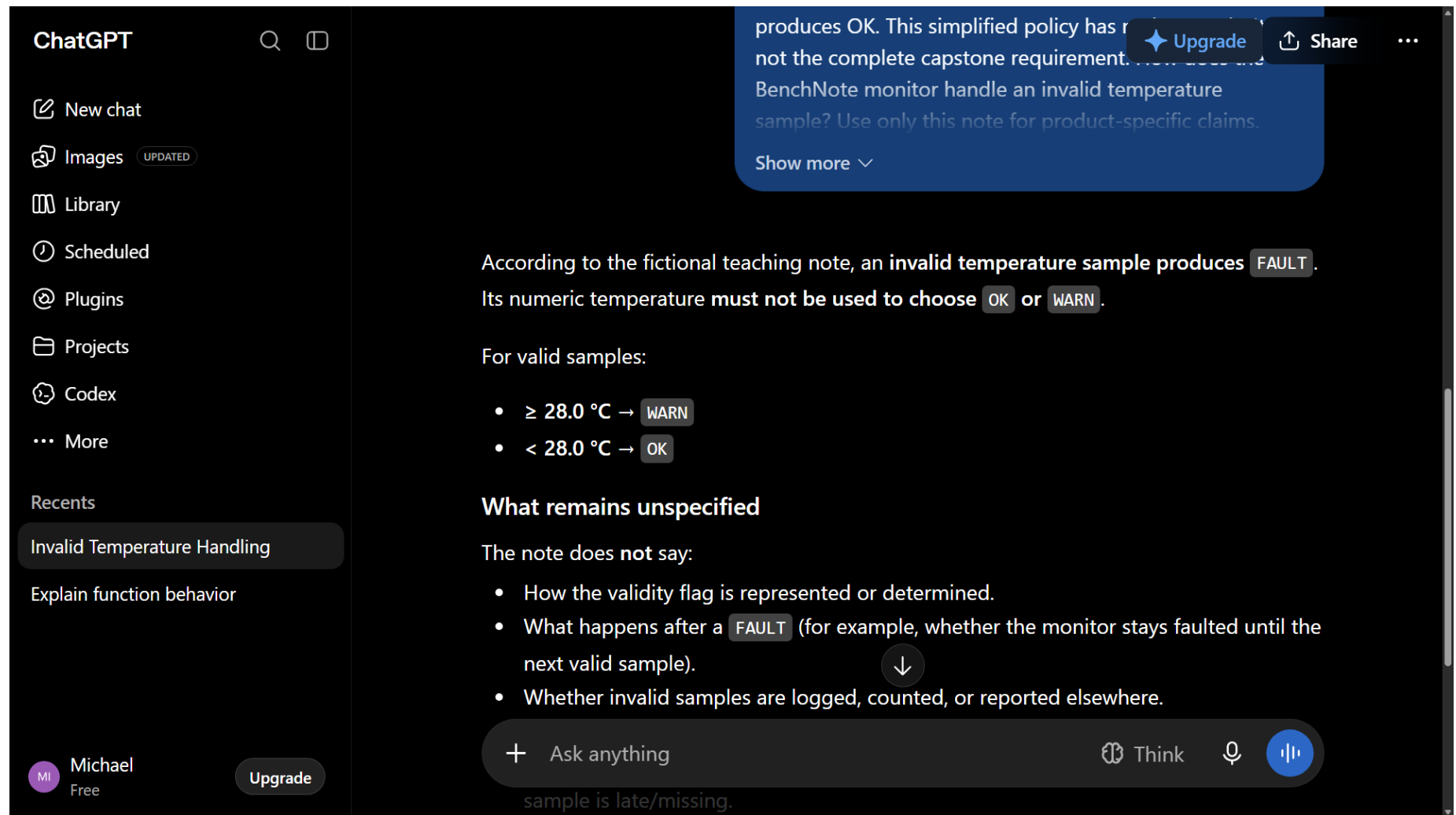
- If `temperature_c >= 28.0°C`, it returns `"WARN"`.
- Otherwise, it returns `"OK"`.

So there is **no validation for invalid values** such as a sensor error, missing reading, `NaN`, or an out-of-range temperature. Those cases would be passed through the same comparison and could therefore produce `"OK"` or `"WARN"` depending on the value and language/runtime behavior.

I couldn't find a public BenchNote monitor specification describing invalid-sample handling; the public BenchNote results I found refer to unrelated workshop-management products. [benchnote.app](#) +1

+ Ask anything Think

SS02-01. Without the teaching note, the response conditionally references prior book code and lacks the BenchNote specification.



ChatGPT 🔍 📄

New chat

Images UPDATED

Library

Scheduled

Plugins

Projects

Codex

... More

Recents

Invalid Temperature Handling

Explain function behavior

Michael Free **Upgrade**

produces OK. This simplified policy has not the complete capstone requirement. How does the BenchNote monitor handle an invalid temperature sample? Use only this note for product-specific claims.

Upgrade **Share** ...

Show more ▾

According to the fictional teaching note, an **invalid temperature sample produces** **FAULT**. Its numeric temperature **must not be used to choose** **OK** or **WARN**.

For valid samples:

- $\geq 28.0\text{ }^{\circ}\text{C}$ → **WARN**
- $< 28.0\text{ }^{\circ}\text{C}$ → **OK**

What remains unspecified

The note does **not** say:

- How the validity flag is represented or determined.
- What happens after a **FAULT** (for example, whether the monitor stays faulted until the next valid sample).
- Whether invalid samples are logged, counted, or reported elsewhere.

↓

+ Ask anything Think 🗣️

sample is late/missing.

SS02-02. With the fictional teaching note supplied, the response prioritizes invalidity and distinguishes unspecified recovery behavior.

Restricted Mode is intended for safe code browsing. Trust this folder to enable all features. [Manage](#) [Learn More](#)

resources > m01-l02 > b0.md

Context comparison — actual instructor run

Condition	Actual result
Before note	Conditional reference to earlier code
Supplied paragraph	Invalid → FAULT; valid boundary retained
Same paragraph repeated	Same governing policy facts
Recovery question	Explicitly unspecified

New chat did not establish empty application context. These observed answers do not measure general reliability.
Source: preserved actual Chapter 2 conversations.

< Restricted Mode 0 0 0 Screen Reader Optimized

SS02-03. The actual context comparison preserves the policy facts across repeated prompts and leaves the missing recovery count unspecified.

7. Lab LAB02: repeat and compare

Run the with-reference request again in a fresh conversation, or compare the two saved illustrative answers in your course resources if access is unavailable. Use exactly the same original note. Record whether each response states the fault behavior, the unit, the valid boundary, and the unspecified recovery behavior accurately. Separate different wording from different claims.

Now ask a new question: “How many consecutive valid samples are required to recover from FAULT?” The correct judgment is that the supplied introductory note does not say. Your deliverable is a comparison worksheet plus a proposed follow-up question for the requirement owner. Do not fill the gap with a common design pattern merely to complete the table.

Expected observations include at least one grounded claim and one explicit unresolved item. If the model invents a recovery count, annotate it as unsupported and point to the absence of such a rule. You have succeeded by detecting that problem. If both answers are accurate, explain why that does not prove universal reliability: you tested a small set of requests against one supplied note.

8. Failure and recovery: the convincing wrong answer

Suppose the assistant says the monitor holds its last state on invalid input, then provides an elegant justification about avoiding alarm chatter. That might be a defensible alternative design, but it contradicts the supplied exercise. Mark the statement against the sentence requiring FAULT. Ask for a corrected answer based on that sentence and for the proposed alternative to be labeled separately.

The recovery is complete when your saved artifact matches the source and preserves any decision still needing an owner. Do not silently alter the source to agree with the answer. When a requirement deserves reconsideration, record a proposed change and its tradeoffs. Engineering judgment includes changing requirements deliberately, but not accidentally through repeated paraphrase.

Completion checklist

- ☐ I can explain tokens, context, and inference without claiming access to hidden reasoning.
- ☐ I saved or examined answers with and without the original note.
- ☐ I compared claims rather than wording and identified missing information.
- ☐ I corrected a contradiction by referring to an exact requirement.

Review questions with answers

1. Does a larger context guarantee a better answer? No. Relevant information helps, but extra unrelated or contradictory material can complicate interpretation. Supply a clear, bounded packet and verify its use.

2. Does pasting a datasheet into chat retrain the model? It provides information for the current interaction. That is different from training model parameters. Keep lasting project facts in versioned documents and supply them again when needed.

3. Two answers agree on an undocumented pin. Is it verified? No. Agreement adds no primary reference or measurement. Locate the exact board documentation and inspect the relevant label before relying on the value.

Transfer exercise

Explain to another maker, in fewer than 150 words, why a chat is not a database of verified project facts. Include one example of helpful supplied context and one example of a claim that still needs a test. Your explanation should make the workflow more understandable without dismissing useful assistance.

3. Give AI an Engineering Brief

By the end of this chapter you will have a reusable task-brief template and a completed brief with a result you can evaluate. Start with Chapter 2, checkpoint CP00, your evidence annotation, and this chapter's original [reporting fixture](#). You will ask for a small reporting change, assess the proposed response, and recover from a request that expands beyond its purpose. No hardware or repository history is required yet.

“Fix my sensor” feels specific when you are staring at the troublesome device. To an assistant it omits nearly everything needed to choose a useful action. Which sensor and board? What does the program currently do? What should it do? Does the failure occur during compilation, parsing a fixture, or physical operation? A brief makes the missing context visible and defines when the task is finished.

Writing the brief is engineering work in its own right. If you cannot describe the expected result, an assistant cannot reliably decide whether a change improves it. The purpose is not to discover a universally powerful prompt. It is to turn an intention into an observable, limited task that a person or tool can execute and review.

1. Begin with the goal and current state

Write the desired behavior in one sentence. Then describe the current behavior separately. For the running project, a suitable goal is: “Make the text report identify its input as synthetic so readers cannot mistake a software replay for a physical measurement.” The current state might be: “The sample report contains a temperature and state but no input-origin label.” This distinction prevents the assistant from treating the existing behavior as the requirement.

Include the current project checkpoint or file version. A filename alone does not identify its contents after several rounds of editing. Later we will use a Git commit identifier; now write the checkpoint name and retain an unchanged starter copy. If you have already modified the file, say so and provide the relevant current text. A patch aimed at yesterday's source may be logically sensible and still fail to apply correctly today.

State what has actually been checked. “The supplied sample log contains two records” is inspectable. “The sensor is broken” is a diagnosis that may not have been established. Prefer “The replay reports FAULT on the second record; this is synthetic input; no board is connected.” That description lets the assistant reason about the right system boundary.

Do not omit a failed attempt when it changes the current state. “I tried a new label but did not save the file” differs from “The label edit is saved and the parser now rejects the line.” Preserve the relevant error text and identify which command produced it. A brief is most useful when another person could start from it without reconstructing a long conversation.

2. Supply exact hardware and software context

Context should answer questions that could change the solution. The operating system and shell affect command syntax. The compiler and language mode affect supported constructs. The target and board options affect cross-compilation. The input format affects parsing. A physical sensor's family name is insufficient for board pin decisions; the exact variant and source revision matter.

For an early text exercise, avoid collecting irrelevant details merely because a template has room. If the task is to annotate a supplied response, the serial port is irrelevant. If the task is to build firmware, the board configuration is essential. Scope the context to the decision and link to longer records when needed. Later chapters will make those records durable project files.

The course distinguishes three execution environments. The **authoring environment** produces this material. The **learner workbench** runs the editor and host checks. The **target** is the documented embedded board for cross-compilation and optional physical deployment. A host replay result comes from the learner workbench; it is not a result from the target merely because it exercises shared source.

Write paths in relation to a named root. For example: “From the extracted sensor-monitor folder, inspect fixtures/sample-log.jsonl.” CP00 is an inspection packet: its log uses NORMAL/ALERT and does not supply a validity flag. The separate original reporting fixture in this chapter adds explicit validity for the following exercise; do not infer that field from CP00. Avoid “open the log” when several logs exist. In PowerShell, the current location determines what a relative path identifies. Chapter 4 teaches the commands that confirm that location before executing a build.

3. Set constraints and excluded changes

A constraint tells the assistant what a valid solution must preserve. For the reporting exercise, preserve numeric values, state names, timestamps, record order, and all policy behavior. Permit only an explicit origin label in the explanatory report. Exclude sensor acquisition, dependency updates, timing changes, and unrelated formatting. These limits make review practical.

Good boundaries are grounded in the task. A change to a text label should not replace a sensor library because the replacement creates installation and behavior questions unrelated to the goal. If the assistant believes a forbidden change is necessary, ask it to explain the dependency before editing. That gives you a decision to make rather than a surprise to discover in a large diff.

Separate workflow preferences from enforced permissions. “Inspect only; do not edit files” is an instruction. A runtime restriction that prevents writes is an access control. Use both appropriately, but do not assume the wording of a prompt creates a technical barrier. In the editor lesson you will inspect the actual available actions and compare the final file state with the baseline.

Also constrain the kind of claims a response may make. “Report commands actually run separately from proposed commands” is useful. “Do not claim a physical measurement from this fixture” protects the meaning of the result. “List unresolved assumptions” keeps the answer reviewable when information is missing. These are evidence requirements, not requests for theatrical caution.

4. Provide sources and a useful reproducer

A **reproducer** is a small set of inputs and steps that produces the behavior under discussion. For a formatting change, two sample records and a desired report can be enough. For a boundary defect, use inputs immediately below, at, and above the threshold. Choose examples that distinguish a correct implementation from a plausible wrong one.

Here is an original illustrative input for this chapter, expressed as a table to avoid implying an executable log format:

Sample	Origin	Valid	Temperature	Existing report
A	synthetic fixture	yes	27.9 °C	27.9 C OK
B	synthetic fixture	yes	28.0 °C	28.0 C WARN

The desired report adds `source=synthetic` to both lines while preserving the other values. The table is a teaching example, not recorded program output. When you use the actual original reporting fixture, copy its real field names and retain the filename. Do not edit a captured result to fit this simplified illustration.

Supply an original requirement excerpt that explains the label. A good excerpt says why the distinction matters: “Every replay demonstration must identify its readings as synthetic. A replay report must not claim that a physical board was measured.” This does not require reproducing a full manual. For external facts, give the document title, exact section or page, revision, and link. A source URL without the relevant proposition still leaves review work unfinished.

If you shorten a log, state the selection. “Records 1 and 2 from the supplied fixture; no values changed” is better than a mysterious fragment. If you construct sample input, label it synthetic. The same discipline that makes a research claim traceable also makes a debugging request easier to answer.

5. Define the output and acceptance criteria

The **output artifact** is what you want returned: a short explanation, a proposed plan, a patch, a test table, or a revised document. Choose it explicitly. “Explain before editing” and “implement the change” are different requests. For a beginner's first brief, ask for a plan and an example result before granting an agent file-edit work.

Acceptance criteria say how you will judge the artifact. They should be observable and proportionate to the change. For this example: both lines contain the origin label; temperatures

and state labels are preserved; invalid samples remain invalid; no physical result is asserted; and no unrelated file is changed. A sentence such as “make it robust” supplies none of those checks.

Include at least one negative case. If a record is invalid, the report should not turn its numeric field into an asserted room temperature. If the input origin is unknown, the response should not invent a known source. Those cases reveal whether the proposed improvement preserves the evidence distinction under less convenient input.

Do not use the assistant's own paraphrase as the only test of acceptance. Compare the returned artifact with the brief yourself. Later, run deterministic checks where appropriate. For a short writing change, an exact inspection may be sufficient; for a policy change, executable tests provide stronger behavioral evidence. Choose the check that matches the risk and claim.

6. Worked example: rescue “fix my sensor”

The initial request is: “Fix my sensor. The output looks wrong. Clean things up too.” Missing information includes the system boundary, observed symptom, expected result, affected files, and meaning of “clean.” An assistant may respond with a list of guesses or propose a broad rewrite. Neither can establish which change is warranted.

Replace it with the following completed brief. This is an original instructional brief, ready to adapt to your actual checkpoint:

```
Goal: Make the explanatory replay report visibly identify synthetic input.
Start: Chapter 3 original resources, unchanged copy; no physical board connected.
Input: resources/reporting-input.jsonl and Chapter 2 BenchNote note.
Current behavior: the sample report can be read without noticing input origin.
Requested output: a short plan and two illustrative revised report lines.
Constraints: preserve sample values, validity, state names, and record order.
Do not change firmware, timing, dependencies, or the input fixture.
Sources: use the supplied note for this exercise's rules; identify missing facts.
Accept when: origin is explicit, all original facts are preserved, and no
physical measurement or executed-command claim is invented.
First identify any ambiguity that would change the proposed report.
```

Submit it with the actual relevant resource text, or use the course's original reference response when account access is unavailable. A suitable illustrative plan identifies the report surface, adds the label, compares old and new values, and calls out the need to retain invalid status. It does not install a library or guess a board pin.

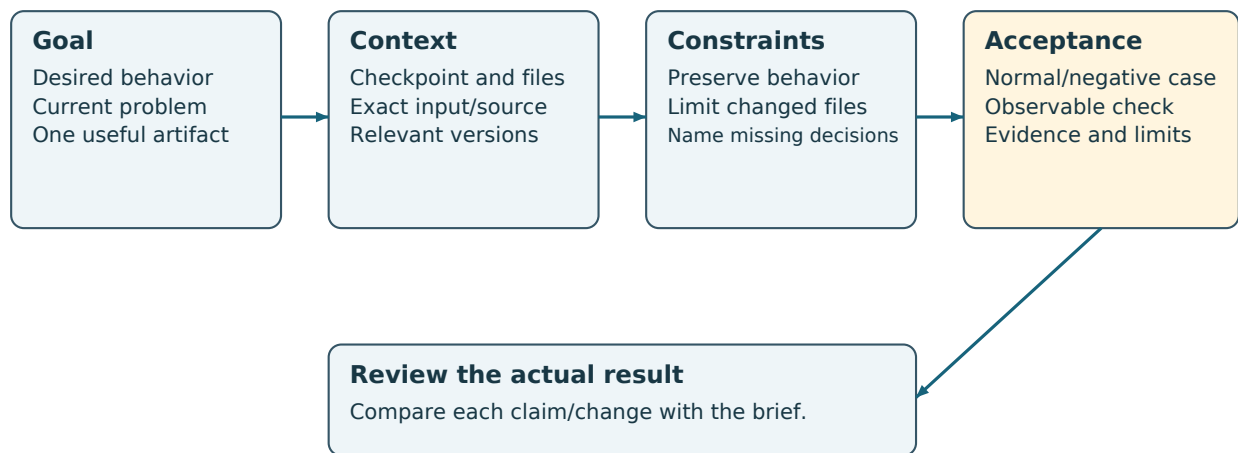
Now assess the response. Suppose it proposes a configurable cloud dashboard “for future scalability.” That is outside the brief, even if it could be useful later. Save the idea in a separate future-work note if you want it, and keep the current task bounded. Suppose it changes WARN to ALARM for readability. That violates the state-name constraint. Ask for a correction tied to that criterion.

A short review note might read: “The proposed origin label meets the goal. Reject the state-name change because downstream examples rely on WARN. No commands were

executed; the shown lines are illustrative. Next step: apply only the approved report edit and inspect the result.” This note captures a decision another person can follow.

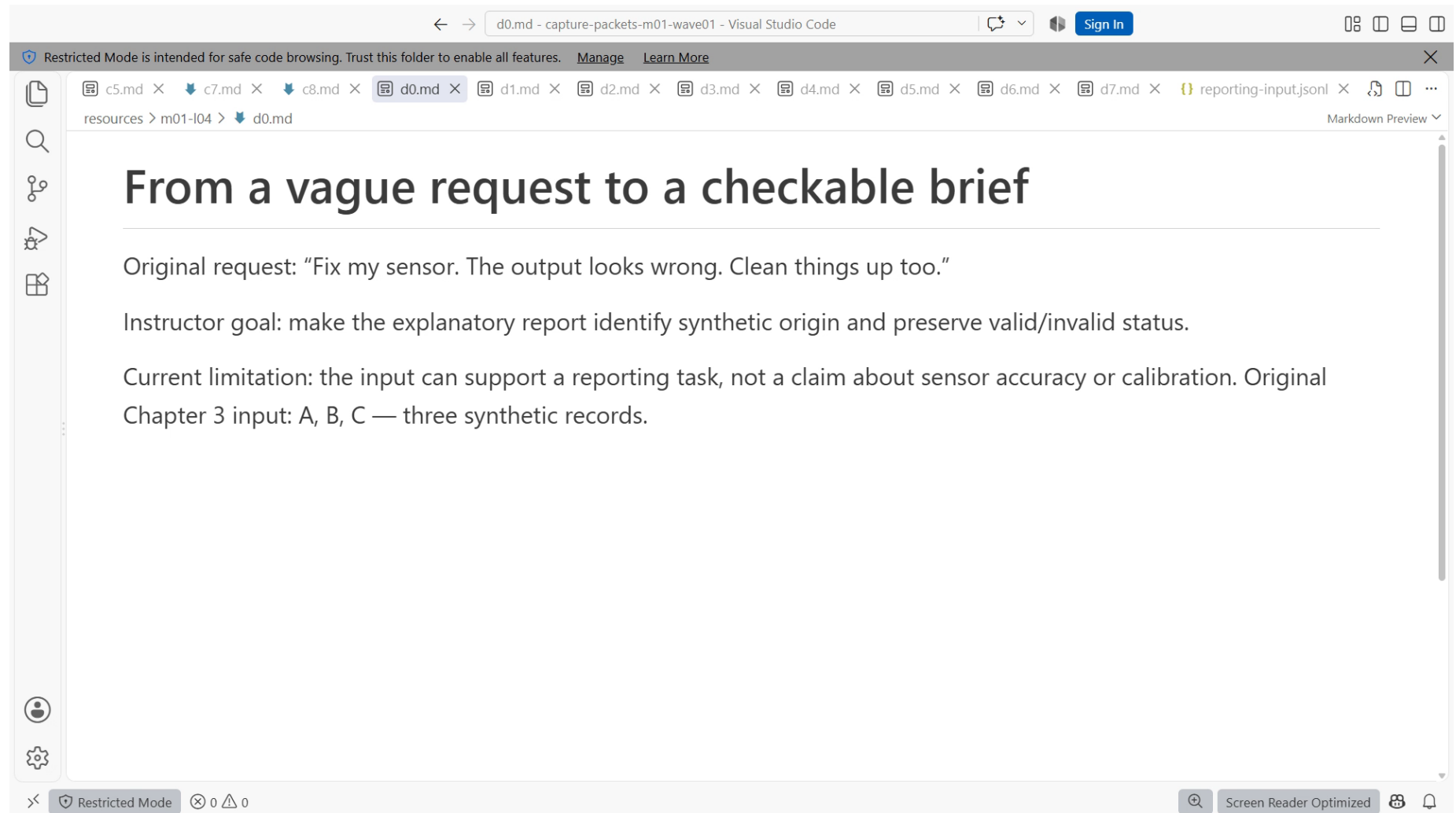
Figures SS03-01 and SS03-02 show the initial and completed briefs. SS03-03 records the actual plan returned during the demonstration and its review. The important difference is visible missing information becoming explicit, not an artificially perfect AI response.

D04 · From intention to an engineering brief

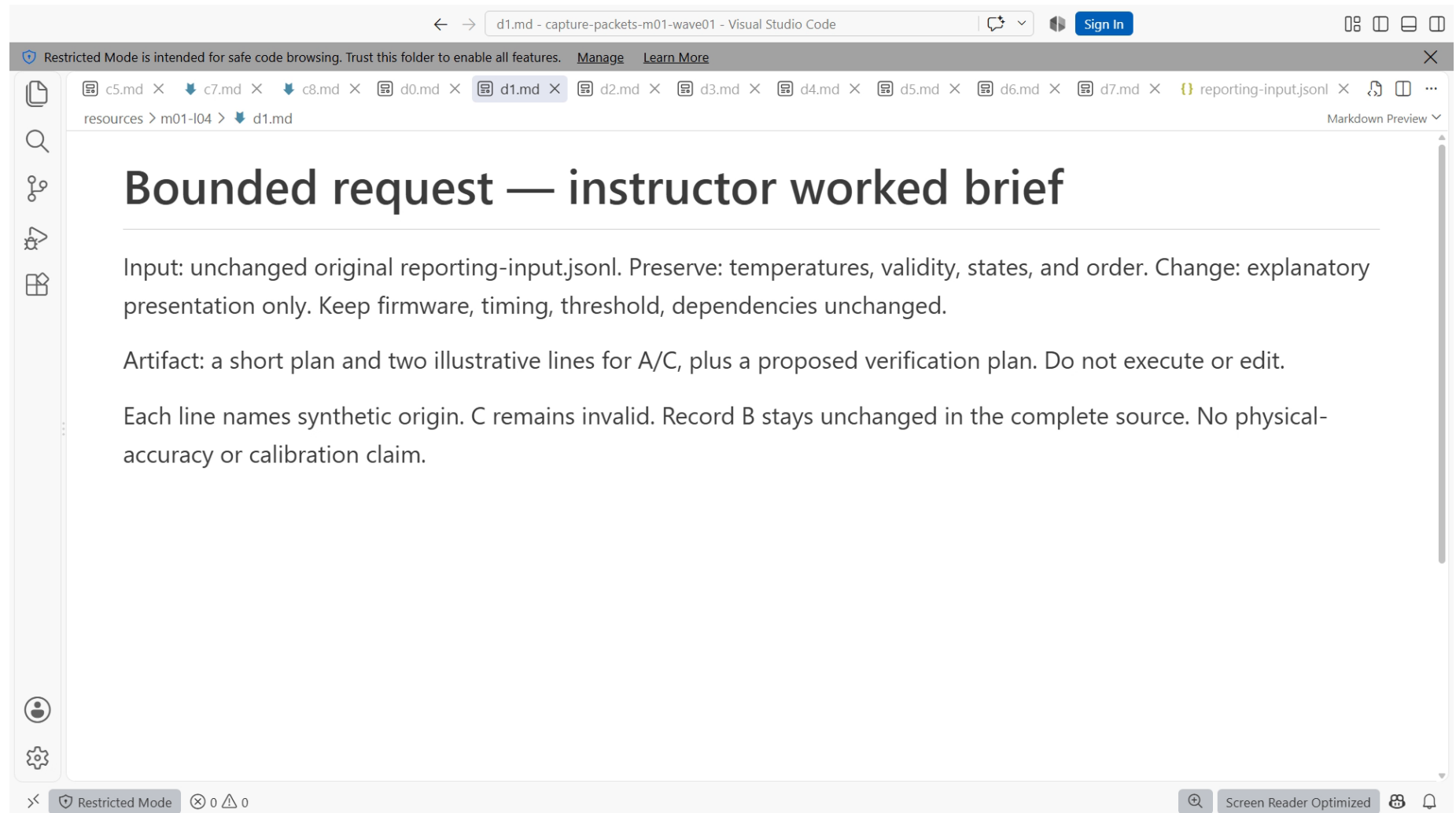


A longer prompt is useful only when its added detail changes or clarifies the decision.

D04. A goal, relevant context, constraints, and observable acceptance produce a reviewable task. The actual result is compared with acceptance.



SS03-01. The initial request leaves acceptance unspecified.



SS03-02. The brief fixes scope and acceptance.

ChatGPT 🔍 📄

New chat

Images UPDATED

Library

Scheduled

Plugins

Projects

Codex

... More

Recents

Revise synthetic report lines

Invalid Sample Handling

Invalid Temperature Handling

Explain function behavior

Michael Free Upgrade

Starting state: Chapter 3 original reporting resource, reporting-input.jsonl copied without changes. This three-record BenchNote exercise is distinct from CP00's

Show more ▾

Upgrade Share ...

No additional information is needed. The supplied JSONL is sufficient and will be treated as authoritative.

Short plan

1. Keep the three input records unchanged, including B and the original order.
2. Revise only the explanatory presentation of A and C so each line explicitly identifies the data as synthetic.
3. Preserve validity as authoritative: C's 31.0 °C does not override valid:false.
4. Verify the proposed output against the stated acceptance criteria without making any physical-accuracy or calibration claim.

Illustrative revised report lines

Illustrative report lines

+ Ask anything Think 🗣️ 🎧 🎵

Sample A — Synthetic data, temperature 27.5 °C, valid: true, state: OK

SS03-03. The actual bounded plan preserves all input records, synthetic origin and the false validity flag.

7. Lab LAB03: specify a reporting change

Your independent task is to request an explanatory summary that counts valid and invalid records in this chapter's original `resources/reporting-input.jsonl` while preserving the original input. Open the actual fixture, identify the records, and count them yourself before asking for help. It contains three original synthetic records with an explicit validity flag; CP00's different two-record log does not define validity. Do not assume a count from this chapter's illustrative two-row table; your release resource is authoritative for the lab.

Create `task-brief.md` using the worked template. Name the input file, request the summary format, forbid rewriting the fixture, and require the response to distinguish a proposed command from an executed result. Add acceptance criteria for the total count, the valid/invalid split, and the synthetic origin label. Supply the small relevant input to the assistant or analyze the provided reference response.

Save a review beside the brief. Accept or reject each criterion with a reason. Expected observations are a bounded plan or summary, explicit synthetic labeling, and counts traceable to the actual fixture. If a response counts incorrectly, you can still pass the lab by detecting the mismatch and correcting the artifact. The deliverable is a sound brief and justified review, not a record that an assistant was always right.

8. Failure and recovery: unrelated rewrites

If an agent changes several unrelated files, stop further edits and inspect the state. Preserve the candidate result long enough to understand it. Return to the unchanged starter copy for this early lab, or use the later Git recovery techniques when history exists. Do not paste a blanket deletion command into an unfamiliar folder.

Rewrite the brief with the exact permitted file, requested output, and success checks. Ask for a small plan before implementation when the dependency is still uncertain. A restart is successful when the new result is reviewable and the original material remains recoverable. Repeatedly adding “be careful” without clarifying the missing boundary is unlikely to solve the real problem.

Completion checklist

- [] My brief separates goal, current state, input, constraints, and requested artifact.
- [] Every path has a named root and every output has an evidence label.
- [] My acceptance criteria include a normal case and a useful negative case.
- [] I reviewed the actual response and preserved the starter material.

Review questions with answers

1. Why provide both expected and actual output? Their difference defines the problem. Actual output alone may be valid under a different requirement, while expected output alone hides the symptom and reproducer.

2. Should every brief request a long plan? No. A small, well-understood edit may need only a few steps. Planning is valuable when it exposes ambiguity, dependencies, or scope before a change; length is not the measure of quality.

3. Is asking another model to approve the answer sufficient verification? No. Another model can help find issues, but acceptance still depends on evidence such as a source, inspected diff, independent count, or executed test.

Transfer exercise

Expand the five-line note from Chapter 1 into a complete brief for your own project. Give it to another person, or review it after a break without consulting your original thoughts. Identify the first question they would need answered and improve the brief. You are ready for the workbench when the task has a clear starting state and a result you can recognize.

4. Set Up Your AI Workbench

You will finish this chapter with a named project folder, a recorded toolchain, a baseline software check, and an explanation of what your agent can access. Begin with the Chapter 3 brief and a copy of checkpoint CP00. Keep authentication private. The core path uses an editor, Git, a host compiler, and Arduino CLI; no physical board or separately billed API is required.

Setup succeeds when you can inspect and reproduce work in a known place. A row of installed icons is not enough. We will connect each tool to a small action: the editor displays source, PowerShell shows the current folder, Git identifies repository state, the compiler checks code, and the agent reports which files it actually used. When something fails, these boundaries help locate the problem.

1. Understand accounts, usage, and privacy

Browser chat, a coding agent signed into a subscription, and an API are different access paths. The course uses the browser and the selected Codex subscription sign-in path. It does not require putting an API key in the project or enabling separately billed usage. Available features and limits can change; check the current official account information and the controls actually shown to you. The edition's pilot verified a dedicated Free account could perform the bounded Codex task; that observation does not promise unlimited course access. [Codex pricing and access](#)

If you reach a limit, save your files and use the original reference responses included with the exercises. You can still practice claim annotation, write briefs, read diffs, and run local checks. Record which AI interactions you performed and which supplied examples you analyzed. Do not label an offline analysis as a new model run. Resume the agent exercise when the included access is available.

Before sharing material with any service, inspect what the request includes. A short code excerpt may be sufficient; a whole folder may include unrelated data. Use a practice project whose contents you understand. The recorded course uses a separate demonstration profile and book accounts. You do not need to reproduce the author's production infrastructure, but you should keep credentials, recovery codes, private messages, and unrelated work out of your learning artifacts and screen recordings.

Sign-in screens can require human actions that cannot be prerecorded meaningfully. In the course video, the safe entry screen and post-login project state are shown with the private interval omitted. An omission of credential entry is not a failure to teach setup. The important observable result is that the intended account can use the intended project, without displaying secrets.

2. Install and identify the editor interface

Use the official Visual Studio Code Windows setup page to choose the installer appropriate to your architecture. Windows on ARM and x64 Windows are distinct environments. The validated demonstration environment used the ARM64 User installer so ordinary editing ran under a standard user account. Follow the current installer instructions and restart the editor after installation. [VS Code Windows setup](#)

Open the Extensions view in VS Code, search for the official Codex extension, and verify its publisher against the official installation documentation before installing. Similar names are not proof of identity. Record the installed version in `setup-record.md`. If the interface differs from the screenshot, use the named function rather than assume a button stays in the same position. The Codex documentation is the source for the selected interface and sign-in flow. [Codex IDE extension](#)

The recorded setup for this edition used VS Code 1.137.0 ARM64 and Codex extension 26.908.40401 on Windows 11 ARM. It also used Codex CLI 0.154.0-alpha.6.2 for a separately recorded command-line pilot. An installed extension and a successful CLI task are different observations. Do not claim an editor-agent action occurred merely because the CLI could act in the same project.

Figure SS04-01 shows the official setup page used during the lesson. SS04-02 shows the installed official extension. SS04-03 shows a safe post-authentication project state. These are teaching records tied to the demonstrated version, not instructions to install an obsolete release forever. When using a newer version, record it and rerun the relevant checks.

Visual Studio Code Features Docs Release Notes Blog Learn Events Resources

Search Ctrl+Shift+P Download

Blog: Enable persistent, portable agent sessions with the Agent Host

Download Visual Studio Code

Free and built on open source. AI agents, Git, debugging, and extensions included.



↓ Windows
Windows 10, 11

User Installer [x64](#) [Arm64](#)
System Installer [x64](#) [Arm64](#)
.zip [x64](#) [Arm64](#)
CLI [x64](#) [Arm64](#)



↓ .deb
Debian, Ubuntu

↓ .rpm
Red Hat, Fedora, SUSE

.deb [x64](#) [Arm32](#) [Arm64](#)
.rpm [x64](#) [Arm32](#) [Arm64](#)
.tar.gz [x64](#) [Arm32](#) [Arm64](#)
Snap [Snap Store](#)
CLI [x64](#) [Arm32](#) [Arm64](#)



↓ Mac
macOS 12.0+

.dmg [Intel chip](#) [Apple silicon](#) [Universal](#)
CLI [Intel chip](#) [Apple silicon](#)

By downloading and using Visual Studio Code, you agree to the [license terms](#) and [privacy statement](#).

Want new features sooner?
Get the [Insiders build](#) instead.

Use [vscode.dev](#) for quick edits online!
GitHub, Azure Repos, and local files.

SS04-01. Windows Arm64 user installer option documented.

The screenshot shows the Visual Studio Code interface with the Extensions Marketplace open. The left sidebar displays the 'Extensions: Marketplace' view with a search filter '@id:openai.chatgpt'. The main panel shows the details for the 'Codex - OpenAI's coding agent' extension by OpenAI. The extension is installed and is currently active. The interface includes a top bar with a 'Sign In' button and a bottom status bar with 'Screen Reader Optimized' and a bell icon.

Extension: Codex - OpenAI's coding agent

Codex - OpenAI's coding agent

OpenAI openai.com | 14,248,175 | ★★★★★ (565)

Codex is a coding agent that works with you everywhere you code — included in ChatGPT Plus, Pro, Business, Edu, and Enterprise plans.

[Switch to Pre-Release Version](#) [Disable](#) [Uninstall](#) ☒ Auto Update [Settings](#)

DETAILS **FEATURES**

Codex is OpenAI's coding agent that helps you write, review, and ship code faster. Use it side-by-side in your IDE or delegate larger tasks to the cloud.

Pair with Codex

Add Codex as a panel in VS Code to chat, edit, and preview changes seamlessly. With context from opened files and selected code, you can write shorter prompts and get faster, more relevant results. (Tip: many users like moving the Codex panel to the right side of the editor.)

Delegate to Codex in the cloud

Offload bigger jobs to Codex in the cloud, then track progress and review results without leaving your IDE. For finishing touches, open cloud tasks locally. Codex keeps the context consistent.

Sign in with your ChatGPT account

Codex works with Plus, Pro, Business, Edu, and Enterprise ChatGPT plans. Learn what's included in your plan at the [Help Center](#).

Installation

Identifier	openai.chatgpt
Version	26.908.40401
Last Updated	13 hours ago
Size	907.34MB

Marketplace

Published	2 years ago
Last Released	2 days ago

Categories

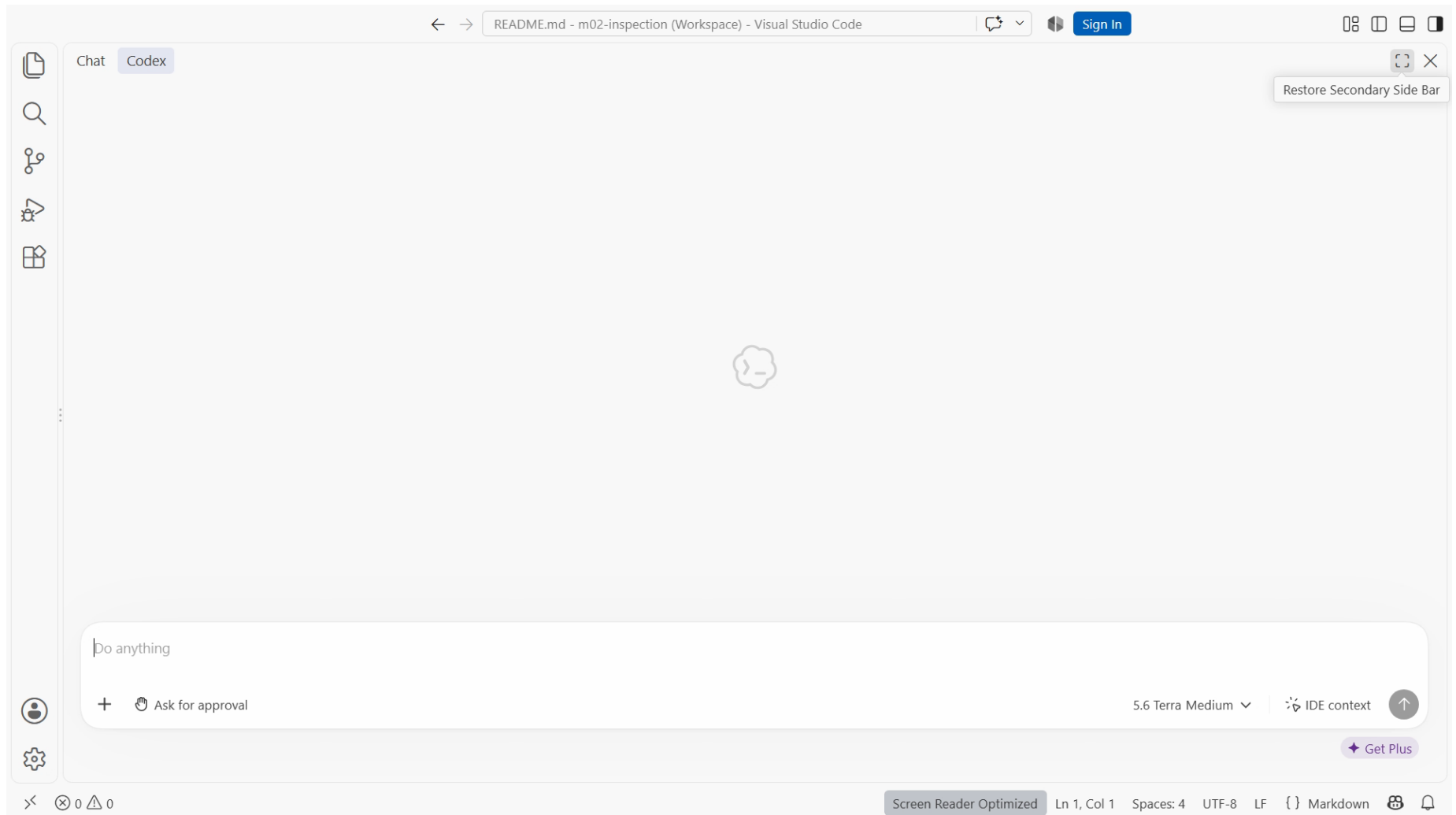
[AI](#) [Chat](#)

Resources

- [License](#)
- [OpenAI](#)

Screen Reader Optimized

SS04-02. Verify the installed extension and publisher.



SS04-03. Inspect the neutral project composer first.

3. Learn the folder and terminal boundary

A **terminal** is an interface for entering commands and reading output. **PowerShell** is the shell used for the Windows examples. The **working directory** is the folder relative paths start from. Opening the right file in an editor does not guarantee that a terminal is in the same folder. Check before running commands that read or change a project.

Create a learning folder in a location you control, such as C:\BookLab, and extract the checkpoint into C:\BookLab\sensor-monitor. The path is an example; choose an equivalent local folder if necessary and substitute it consistently. Use VS Code's Open Folder command to open that exact sensor-monitor directory. Its Explorer should show the supplied resources and README, rather than a parent directory containing unrelated projects.

In VS Code, open a new integrated PowerShell terminal. Run these commands one line at a time:

```
# Context: PowerShell in the intended learner project.
Get-Location
Get-ChildItem
```

Get-Location prints the current folder; Get-ChildItem lists its contents. If the folder is wrong, change it explicitly:

```
# Context: PowerShell; substitute your actual extraction path.
Set-Location 'C:\BookLab\sensor-monitor'
Get-Location
```

Quotes keep a path with spaces together. The . in a path means the current directory. A path beginning .. moves to the parent. In the coming commands, .\scripts\replay.cmd names the Windows replay wrapper inside the current project's scripts folder. Keep paths intact when copying them from a page; a visual line wrap is not a new command.

Keep a terminal result with the command that produced it. A message saying “file not found” is hard to interpret without the working directory and requested path. Do not try random folders until the error disappears. Compare the README's expected tree with your extraction and decide which mismatch explains the failure.

4. Record the minimal toolchain

Install Git for Windows from its official distribution and follow the project README for the host compiler and Arduino CLI. Arduino CLI's documented installation options include downloadable binaries; the board platform and libraries are additional dependencies. A successful arduino-cli version does not establish that the selected ESP32 target is installed. [Git installation](#), [Arduino CLI installation](#)

The bounded Windows ARM rehearsal used Arduino CLI 1.5.1, ESP32 core 3.3.11, and native ARM64 Clang 23.1.1. It compiled the pilot and executed its synthetic policy checks under the standard user. That validates the recorded combination for that exercise. The complete

reference project has its own dependency record and release checks; follow its `dependencies.json` and `README` when building the full project.

Run the available version commands in PowerShell and copy their actual output into your record:

```
# Context: PowerShell in sensor-monitor; these commands inspect installed tools.
git --version
arduino-cli version
clang++ --version
```

If the project selects a different supported host compiler, record that compiler and its command instead. Do not rename an executable or copy it into a system folder merely to make the example command work. Follow the documented installation path, then open a fresh terminal so it receives updated environment settings.

Your setup record should name date, operating system and architecture, editor and extension, shell, compiler, Arduino CLI, board core, selected checkpoint, and commands run. Add result and limitation columns. “Compiler starts” is one result; “the CP01 host replay runs” is another; “firmware cross-build passes” is another. Recording them separately turns a future failure into a comparison with a known baseline.

5. Compile the supplied baseline without hardware

Open the reference project's `README` before running its scripts. CP00 is an inspection packet and needs no build. Use a separately extracted CP01 baseline for the executable setup check. From that CP01 project root, the primary Windows entry points are:

```
# Context: PowerShell in the extracted sensor-monitor project root.
.\scripts\replay.cmd
.\scripts\build-firmware.cmd
```

Run them individually and inspect each result before continuing. The first compiles and executes the minimal host report, which prints a fixed **synthetic** value. The second cross-compiles the baseline for the documented target. CP01 has no fixture parser or automated assertions; those arrive later. These `.cmd` wrappers require no PowerShell execution-policy change and manage the supported native runtime path inside their own process. Follow the matching dependency and setup records.

The expected observations are a successful host replay exit status, the fixed synthetic JSON sample documented in CP01, and a successful cross-build with generated artifacts. These are **expected checks for your run**, not claims that you have already performed them. Record actual output, including a failure. Do not copy the book's expected label into a result field without execution.

In the separately verified pilot, the deliberately wrong comparison failed only the 28.0 °C case; the corrected comparison passed all three synthetic cases. The pilot firmware cross-build produced 355,240 program bytes and 54,576 global-variable bytes with its recorded options.

Those numbers belong to that pilot, not to every future project build. They do not establish a board upload, electrical operation, or measurement accuracy.

The explicit example target includes options as well as a board name:

```
esp32:esp32:adafruit_feather_esp32s3:CDCCOnBoot=cdc,PSRAM=enabled,FlashSize=4M,CPUFreq=240,FlashMode=qio,PartitionScheme=tinyuf2_noota
```

Treat the full target configuration as part of the build input. Copy the value from the matching release instructions rather than reconstruct it from memory. A similarly named board menu selection is not necessarily the same build.

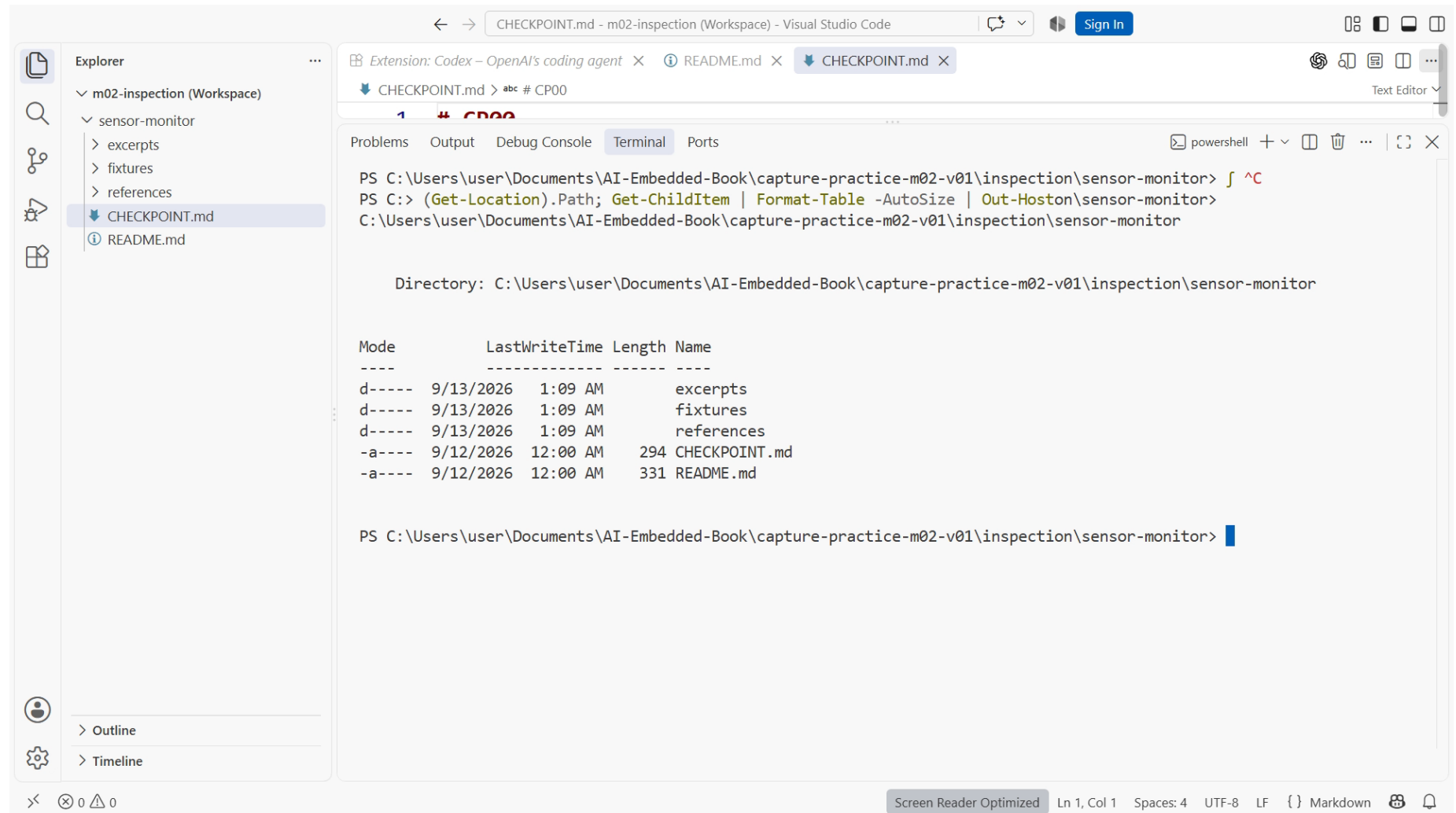
6. Worked example: inspect with bounded access

Open the Codex view and inspect its current access settings. Identify whether it can read files, write inside the project, execute commands, or use the network. The available controls and sandbox behavior vary by version and platform. The Windows documentation describes the selected sandbox choices; a prompt is not a substitute for those controls. [Codex Windows sandbox](#)

The edition's CLI rehearsal used a workspace-write scope with network access disabled and the supported unelevated Windows sandbox. That is a recorded configuration, not an instruction to weaken a machine's security or grant administrator access for everyday edits. Follow the current supported setup and keep your selected project scope narrow.

Submit this inspection brief: “Read the README and the introductory reporting example. Explain the input, output, and current verification limits. Do not edit files or run installation commands. Name every file you use, and separate commands you propose from commands actually run.” Watch the action record, not just the final paragraph. If it requests access beyond the intended task, inspect why before allowing it.

A good result names the actual files and explains the source it read. Compare its description with the open code. If it refers to a missing file or claims a physical test, mark the discrepancy and ask for correction. Figure SS04-04 shows the project folder; SS04-05 shows the relevant access configuration; SS04-06 shows the real inspection actions and result. Capture follows the actual interface behavior.



Visual Studio Code interface showing a workspace named "m02-inspection (Workspace)". The Explorer sidebar shows a folder named "sensor-monitor" containing files "CHECKPOINT.md" and "README.md". The terminal window displays the following PowerShell commands and output:

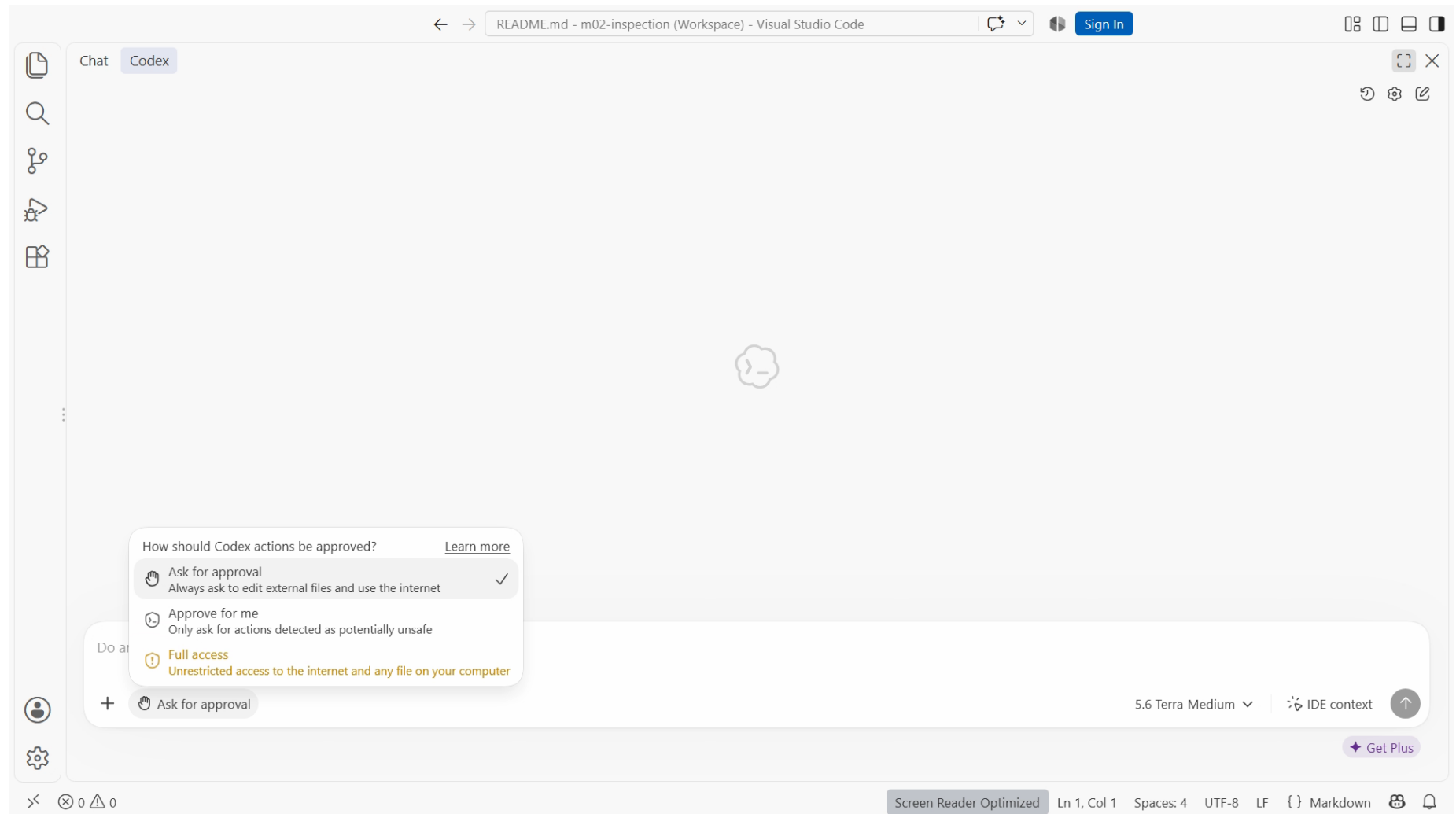
```
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\inspection\sensor-monitor> j ^C
PS C:> (Get-Location).Path; Get-ChildItem | Format-Table -AutoSize | Out-Host\sensor-monitor>
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\inspection\sensor-monitor
```

Directory: C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\inspection\sensor-monitor

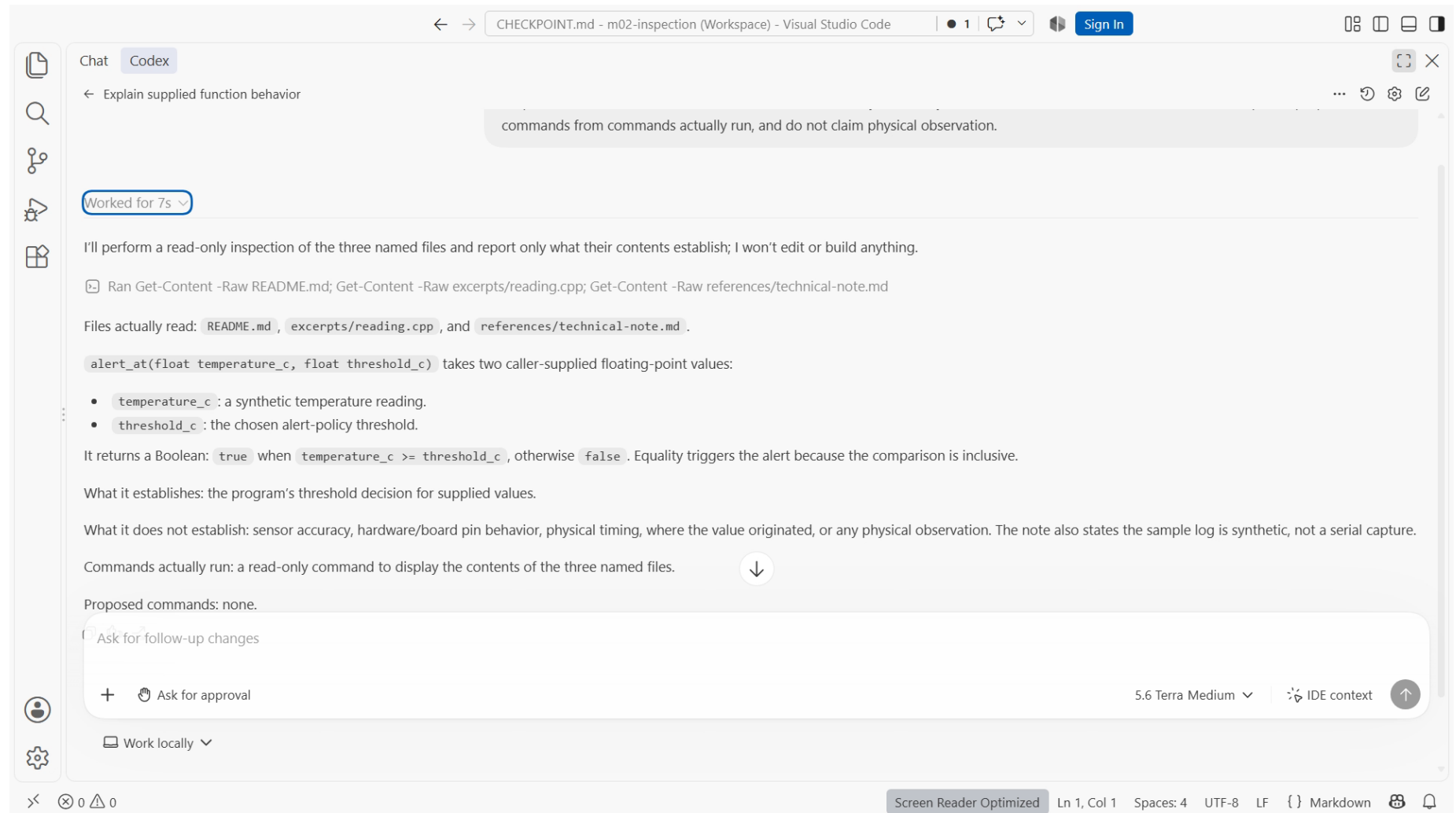
Mode	LastWriteTime	Length	Name
d-----	9/13/2026 1:09 AM		excerpts
d-----	9/13/2026 1:09 AM		fixtures
d-----	9/13/2026 1:09 AM		references
-a----	9/12/2026 12:00 AM	294	CHECKPOINT.md
-a----	9/12/2026 12:00 AM	331	README.md

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\inspection\sensor-monitor>

SS04-04. Actual CP00 path and listing shown.



SS04-05. Read permission choices before submitting work.



Chat Codex

← Explain supplied function behavior

commands from commands actually run, and do not claim physical observation.

Worked for 7s

I'll perform a read-only inspection of the three named files and report only what their contents establish; I won't edit or build anything.

Ran Get-Content -Raw README.md; Get-Content -Raw excerpts/reading.cpp; Get-Content -Raw references/technical-note.md

Files actually read: README.md , excerpts/reading.cpp , and references/technical-note.md .

alert_at(float temperature_c, float threshold_c) takes two caller-supplied floating-point values:

- temperature_c : a synthetic temperature reading.
- threshold_c : the chosen alert-policy threshold.

It returns a Boolean: true when temperature_c >= threshold_c , otherwise false . Equality triggers the alert because the comparison is inclusive.

What it establishes: the program's threshold decision for supplied values.

What it does not establish: sensor accuracy, hardware/board pin behavior, physical timing, where the value originated, or any physical observation. The note also states the sample log is synthetic, not a serial capture.

Commands actually run: a read-only command to display the contents of the three named files.

Proposed commands: none.

Ask for follow-up changes

+ Ask for approval

Work locally

5.6 Terra Medium IDE context

Screen Reader Optimized Ln 1, Col 1 Spaces: 4 UTF-8 LF {} Markdown

SS04-06. Completed CP00 IDE task record shown.

7. Lab LAB04: create a setup record

Your deliverable is `setup-record.md` and a short inspection review. Record the actual folder, tool versions, checkpoint, one baseline command, and its result. Then request a file-based explanation and identify at least two claims you can check in the files. State where a proposed command would run and whether it actually ran.

For success, the record must let you distinguish four outcomes: the tool is unavailable; the tool starts but the project check fails; the project check passes; or an account limit prevents the agent interaction. A learner who documents and correctly diagnoses an unavailable tool has made useful progress, but should leave the baseline check incomplete until it executes successfully. Do not turn an incomplete setup into a pass by borrowing another machine's output.

Stop the agent with the interface's stop control if it is doing the wrong task. After stopping, inspect files and terminal state before resuming. An interrupted turn may have completed some actions. Save the current state and start again with a brief that names what remains. "Stop" is a control action, not a guarantee that nothing happened before it.

8. Failure and recovery

For a wrong-folder error, run `Get-Location` and list the expected root files. Correct the path and rerun only the affected command. For a missing executable, use `Get-Command git` or the relevant tool name to see whether PowerShell can resolve it. Confirm the documented installation, then start a fresh terminal. For a compiler error, preserve the diagnostic and source state; reinstalling the editor is not a targeted remedy.

For an account limit, save the brief and use the supplied reference response while continuing local work. For a failed authentication step, complete the provider's private sign-in process and verify the post-login identity without recording secrets. These failures require different actions. Keeping them distinct prevents a minor path problem from turning into unnecessary environment changes.

Completion checklist

- ☐ I opened the intended folder and confirmed the PowerShell location.
- ☐ I recorded actual installed versions and the selected access path.
- ☐ I ran the applicable baseline check and saved its real result.
- ☐ I inspected an agent action record and checked its file-based claims.
- ☐ I kept synthetic, cross-build, and physical evidence distinct.

Review questions with answers

1. Why can an open file and a failing command refer to different projects? The editor's active document and the terminal's working directory are separate state. A relative path follows the terminal location, so inspect that location before diagnosing source behavior.

2. Does an installed extension prove agent access works? No. Installation, authentication, project access, and a successful action are separate checks. Verify each without inferring one from another.

3. Why keep the full board configuration? Build options can change the compiled target and resource use. Reproduction requires the actual configuration, not just a family name or a screenshot of a similar board menu.

Transfer exercise

Open a second small project you own. Without editing it, record its root, toolchain entry point, and one check you would run. Write an inspection-only brief that names the correct files. Compare this deliberate setup with the assumptions you previously made when asking for help.

5. Git Without the Mystery

This chapter teaches you to inspect a meaningful change and save it as a commit. Begin with a working Chapter 4 toolchain and the CP01 source bundle supplied without learner Git history. Work in a new practice folder. By the end you will have two explanatory commits and will be able to say exactly which contents each saved.

An assistant can make a useful edit quickly. The next problem is knowing what changed and deciding whether to keep it. A copied folder named `final-really-final` does not make that decision clear. Git records selected versions and helps you compare them. We will use a small reporting-label change so the mechanics stay visible while the program remains understandable.

1. Local Git, hosting, and synchronization

Git is version-control software. A local repository contains recorded history and metadata alongside a working copy of project files. **GitHub** is a hosting and collaboration service that can store a remote copy of that history. You can inspect changes and make local commits without pushing anything to GitHub. Chapter 6 adds the remote step.

Cloud file synchronization serves a different purpose. A synchronizer may copy whichever files currently exist; it does not necessarily give you a reviewed, intentional sequence of project changes. Git also does not automatically preserve every untracked file or unsaved editor buffer. It records what you deliberately include in commits. Use it with ordinary backup practices rather than treating either as a complete replacement for the other.

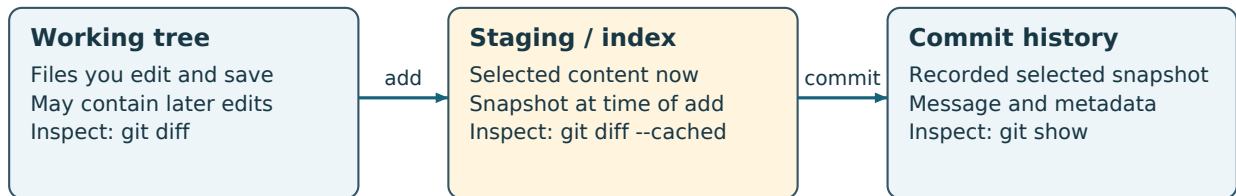
A repository is not a magic “correct” folder. You can commit a broken program, a misleading README, or a secret. Git will faithfully preserve that choice. Our workflow puts inspection and relevant checks before the commit, so history records decisions you can explain. An intentionally failing exercise may be committed too, provided its message and documentation identify it as a seeded fault.

2. Understand the three places

The **working tree** is the project content you edit. The **staging area**, also called the index, is the content selected for the next commit. A **commit** records that selected snapshot with metadata and a relationship to earlier history. Staging a file selects its contents at that moment; later edits can remain unstaged. [Pro Git, recording changes](#)

Figure D05 shows these three places and the operations between them. Saving in the editor changes the working tree. Staging selects content for the next commit. Committing records the staged snapshot. Pushing, which comes later, transfers commits to a remote. Do not call all four actions “saving,” because each answers a different question.

D05 · Three Git states you can inspect



Saving a file does not stage it. Staging does not automatically include later edits.

A clean Git state is not a test result. Pushing is a separate remote operation.

D05. Saving changes the working tree; add selects contents in staging; commit records that selected snapshot. Ordinary and cached diffs compare different states.

Consider a README with a corrected heading. You stage it, then add a troubleshooting paragraph. The staged snapshot has the heading correction, while the working file has both edits. A commit at that point saves the staged version. This is useful when intentional and confusing when accidental. Two diff views will make the distinction visible.

3. Initialize the practice repository

Extract the CP01 source bundle into a new sensor-monitor-git folder, separate from the unchanged starter. The bundle for this exercise must not already contain a .git directory. Open it in VS Code and confirm its README and source tree. Do not run initialization in your home folder, Downloads folder, or the parent of unrelated projects.

In PowerShell, from the extracted practice root:

```
# Context: PowerShell in the new sensor-monitor-git source-only folder.
Get-Location
git init -b main
git status
```

`git init -b main` creates repository metadata and names the initial branch `main`. It does not upload files or create a commit. The expected status lists untracked project files and indicates that no commit exists yet. Exact wording varies with Git version. If Git reports an existing repository, inspect the folder before proceeding; you may have opened the wrong extraction. [git init reference](#)

Set repository-local identity before the first commit. Use your chosen author name and, if you want GitHub attribution without exposing your private address, the exact no-reply address shown in your GitHub email settings. Do not invent an account-specific address from this example:

```
# Context: PowerShell in this practice repository; replace both example values.
git config user.name 'Your chosen author name'
git config user.email 'YOUR-VERIFIED-NOREPLY-ADDRESS'
git config --get user.name
git config --get user.email
```

These are local settings because `--global` is omitted. Commit identity is metadata; it is not GitHub authentication. The host's email privacy settings explain how its no-reply address works. Check your actual setting before saving it. [GitHub commit email](#)

4. Decide what belongs in history

Source code, reproducible scripts, requirements, tests, and useful documentation belong in this practice repository. Generated compiler output can usually be recreated and should be excluded. The supplied `.gitignore` defines relevant exclusions such as the project's build directory. Read it rather than replace it with a long list copied from an unrelated stack.

A `.gitignore` entry prevents matching untracked files from appearing as ordinary candidates for addition. It does not remove a file already tracked in history. It also does not encrypt a credential or make an uploaded secret private. This chapter's deliberate mistake uses harmless generated output after a baseline commit; do not practice with sensitive files. [gitignore reference](#)

Run `git status` and inspect the list. If you see a personal document, credential file, or an entire unrelated directory, stop and correct the project boundary. If you see expected firmware, host code, scripts, and documentation, stage those named paths. The source bundle may contain additional tracked support files; use its README and release tree to account for them.

```
# Context: PowerShell in the CP01 practice repository.
git add README.md .gitignore dependencies.json firmware host scripts fixtures docs CHECKPOINT.md
git status
git diff --cached --stat
git diff --cached
```

Add any other intended source-bundle documents explicitly after reviewing them. Do not use a broad command merely to silence untracked status. In VS Code, the Source Control view shows changed files; its stage control selects them, and the Staged Changes group corresponds to the staging area. Read the diff in either interface before committing.

5. Commit the baseline

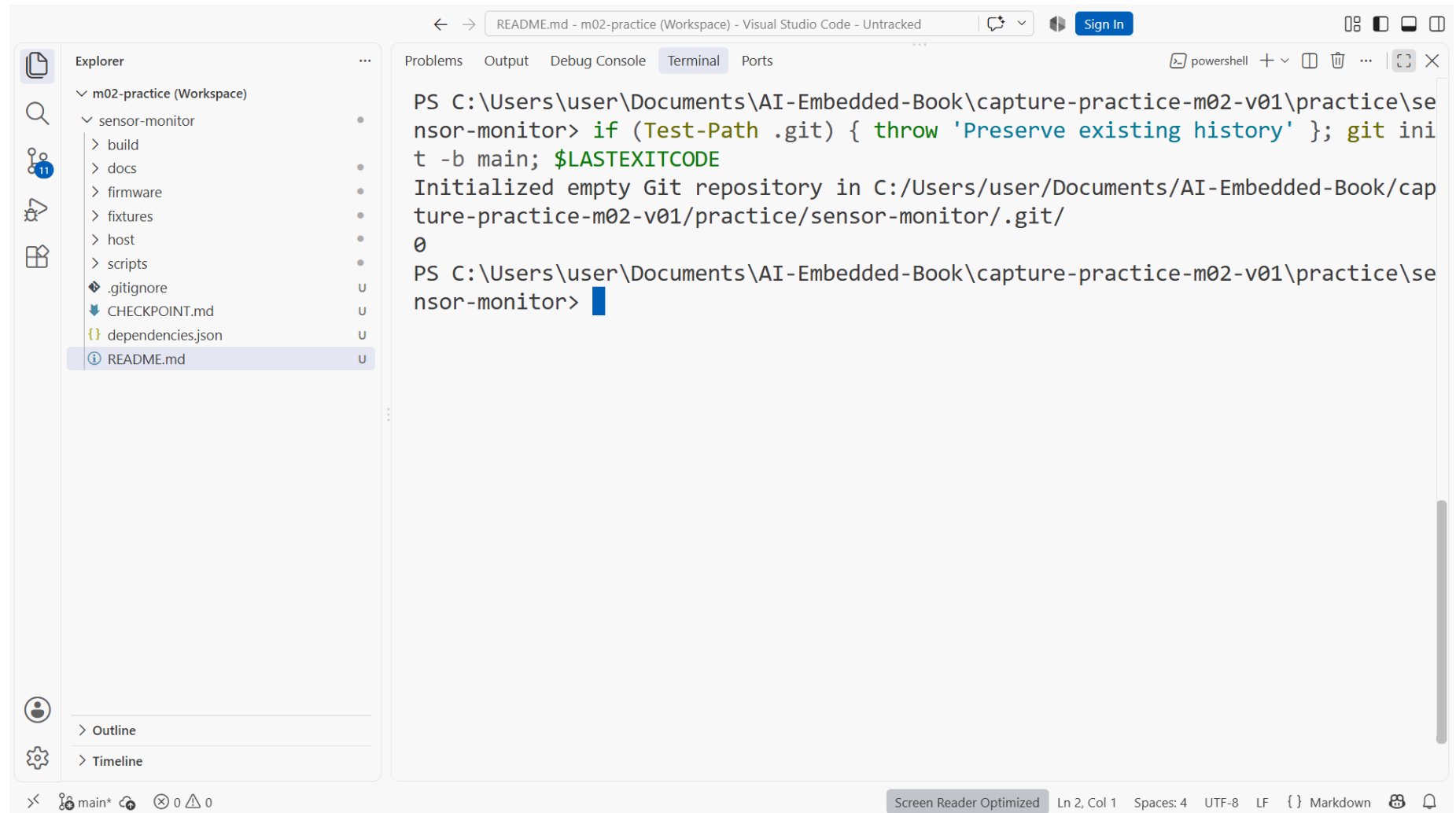
Run the documented baseline check from the practice folder. Save the result in your learner notebook outside generated build output, or in a deliberate evidence file if the exercise calls for it. Then make the baseline commit:

```
# Context: PowerShell in the CP01 practice repository after baseline review.
git commit -m 'Record sensor monitor baseline and build instructions'
git log --oneline -2
git status
```

A commit records the staged content, along with its message and metadata. A useful message explains the purpose of the change. “Update” and “AI changes” do little to help your future self choose a recovery point. A commit's identifier is computed from its contents and metadata, so your identifier will differ from a screenshot's even when the teaching steps are equivalent. [git commit reference](#)

Expected observations are one baseline entry in history and no unintended changes left in the working tree. A clean status does not mean the code is correct; it means the relevant tracked states agree and there are no listed untracked candidates. Keep the test result separate from the Git-state observation.

Figures SS05-01 and SS05-02 show initialization and the reviewed file selection. They should show the actual practice path and expected exclusions, not another repository whose contents happen to look similar. Your own outputs are valid even when line counts and commit IDs differ.

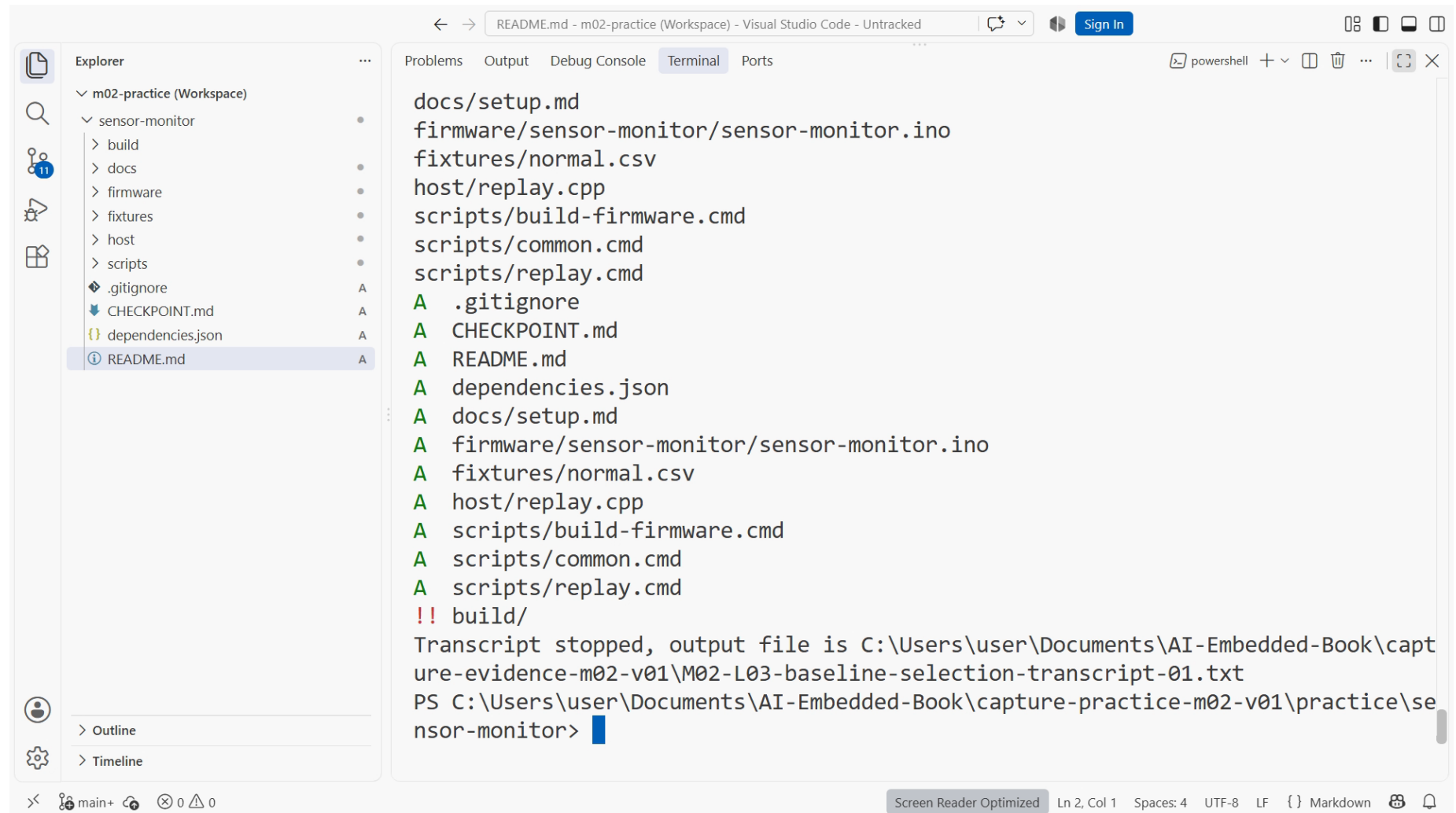


The screenshot shows the Visual Studio Code interface. On the left, the Explorer panel displays the file structure of the 'm02-practice' workspace, with 'README.md' selected. The main editor area shows a terminal window with the following output:

```
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\practice\sensor-monitor> if (Test-Path .git) { throw 'Preserve existing history' }; git init -b main; $LASTEXITCODE
Initialized empty Git repository in C:/Users/user/Documents/AI-Embedded-Book/capture-practice-m02-v01/practice/sensor-monitor/.git/
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\practice\sensor-monitor>
```

The status bar at the bottom indicates the current file is 'main', the editor is 'Screen Reader Optimized', and the cursor is at 'Ln 2, Col 1'.

SS05-01. Git initializes the local practice repository.



Visual Studio Code interface showing a workspace named "m02-practice (Workspace)". The Explorer sidebar on the left lists the following files and folders:

- sensor-monitor
 - build
 - docs
 - firmware
 - fixtures
 - host
 - scripts
- .gitignore
- CHECKPOINT.md
- dependencies.json
- README.md

The Terminal window on the right shows the following output:

```
docs/setup.md
firmware/sensor-monitor/sensor-monitor.ino
fixtures/normal.csv
host/replay.cpp
scripts/build-firmware.cmd
scripts/common.cmd
scripts/replay.cmd
A .gitignore
A CHECKPOINT.md
A README.md
A dependencies.json
A docs/setup.md
A firmware/sensor-monitor/sensor-monitor.ino
A fixtures/normal.csv
A host/replay.cpp
A scripts/build-firmware.cmd
A scripts/common.cmd
A scripts/replay.cmd
!! build/
Transcript stopped, output file is C:\Users\user\Documents\AI-Embedded-Book\capture-evidence-m02-v01\M02-L03-baseline-selection-transcript-01.txt
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\practice\sensor-monitor>
```

SS05-02. Selected source excludes generated build output.

6. Worked example: read a report-label diff

Find the exact line `Report label: synthetic.` in the CP01 README. Change it to `Report label: synthetic input.` and save. This is a human-facing documentation label, not a machine-readable key or firmware edit. The exact path is `README.md`; the baseline's fixed host sample remains 25.00.

The **expected diff** for that edit is:

```
-Report label: synthetic.  
+Report label: synthetic input.
```

This display is the expected comparison, not a captured command result. Inspect your actual diff to establish what changed.

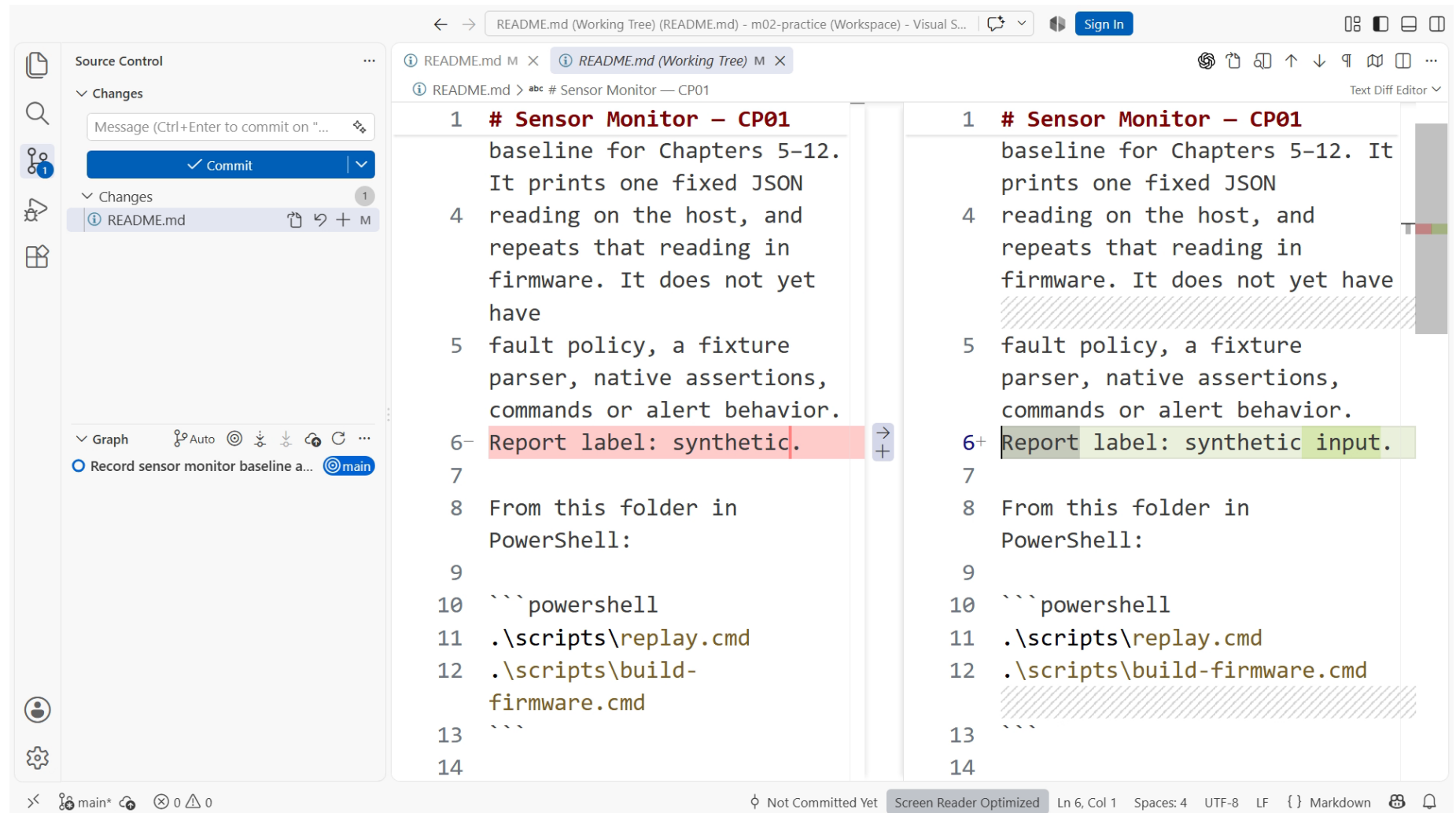
The minus line is the previous content; the plus line is the new content. These markers belong to the diff display and are not characters to paste into the README. Context lines help locate the change. The file header identifies which file changed. A whole file appearing replaced can indicate formatting or line-ending changes that deserve inspection before acceptance. [git diff reference](#)

Run the two comparisons:

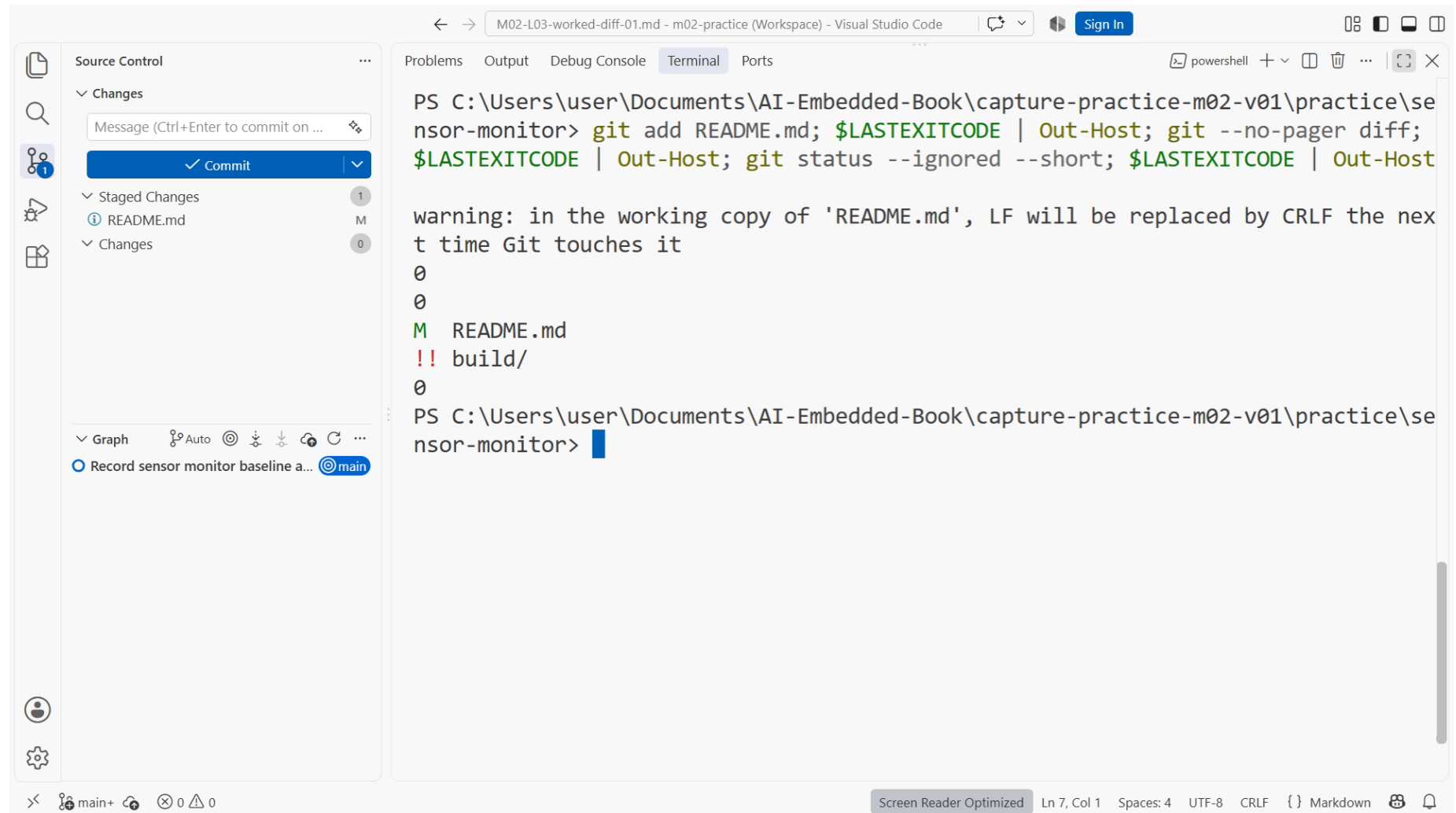
```
# Context: PowerShell in the practice repository after saving the edit.  
git diff  
git diff --cached
```

Before staging, the ordinary diff should show your edit; the cached diff should be empty if nothing is staged. Stage only `README.md`, then run both commands again. The selected edit should move to the cached comparison. This is the moment to explain the change aloud: which label changed, why the input origin is clearer, and which behavior was preserved.

The selected path is the CP01 `README.md`; do not use the expected two-line fragment as proof that your whole working tree is correct. Inspect the actual changed lines, run the relevant project check, and compare the result with the task brief. Figures SS05-03 and SS05-04 belong to this real diff and staged selection.



SS05-03. The diff isolates the reporting-label change.



The screenshot shows the Visual Studio Code interface. On the left, the Source Control sidebar is open, showing a commit message input field with the text "Message (Ctrl+Enter to commit on ...)". Below this, there is a "Commit" button and a list of staged changes, including "README.md". The main editor area displays a PowerShell terminal window with the following commands and output:

```
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\practice\se  
nsor-monitor> git add README.md; $LASTEXITCODE | Out-Host; git --no-pager diff;  
$LASTEXITCODE | Out-Host; git status --ignored --short; $LASTEXITCODE | Out-Host  
  
warning: in the working copy of 'README.md', LF will be replaced by CRLF the nex  
t time Git touches it  
0  
0  
M README.md  
!! build/  
0  
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\practice\se  
nsor-monitor>
```

The terminal window also shows a "Sign In" button in the top right corner. The bottom status bar indicates "Screen Reader Optimized" and "Ln 7, Col 1".

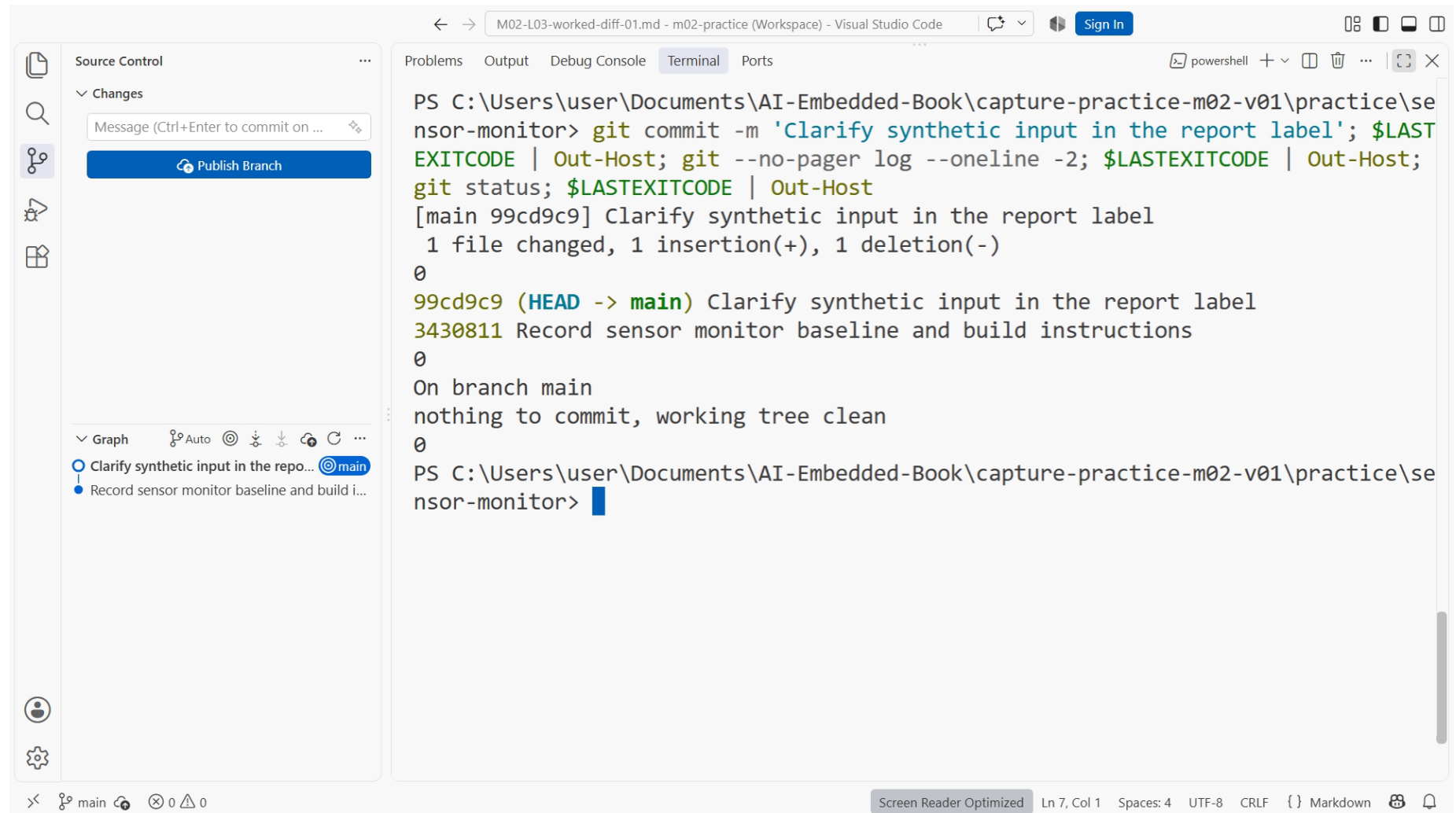
SS05-04. Staging selects the reviewed README change.

7. Lab LAB05: save a meaningful change

Starting from your baseline commit, perform the supplied reporting-label edit independently. Write a one-sentence acceptance statement before editing. Use the editor to make and save the change, inspect the ordinary diff, and run the relevant check. Stage the exact file and inspect the cached diff. Then commit with a message explaining the reporting purpose.

```
# Context: PowerShell in the practice repository; inspect before committing.
git status
git diff --cached
git commit -m 'Clarify synthetic input in the report label'
git log --oneline -2
git status
```

Your deliverable is two explanatory commits plus a short note identifying the baseline and label change. The expected observation is a history whose second entry contains only the intended change and any deliberately associated documentation. Your IDs and timestamps will differ from the book. Figure SS05-05 shows the teaching history and message order.



The screenshot shows the Visual Studio Code interface. On the left is the Source Control panel, displaying a commit message input field with the text "Message (Ctrl+Enter to commit on ...)" and a "Publish Branch" button. Below this is a graph view showing two commits: "Clarify synthetic input in the repo..." (selected, marked as main) and "Record sensor monitor baseline and build i...". The main editor area shows a terminal window with the following output:

```
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\practice\se  
nsor-monitor> git commit -m 'Clarify synthetic input in the report label'; $LAST  
EXITCODE | Out-Host; git --no-pager log --oneline -2; $LASTEXITCODE | Out-Host;  
git status; $LASTEXITCODE | Out-Host  
[main 99cd9c9] Clarify synthetic input in the report label  
 1 file changed, 1 insertion(+), 1 deletion(-)  
0  
99cd9c9 (HEAD -> main) Clarify synthetic input in the report label  
3430811 Record sensor monitor baseline and build instructions  
0  
On branch main  
nothing to commit, working tree clean  
0  
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m02-v01\practice\se  
nsor-monitor>
```

The terminal window title is "M02-L03-worked-diff-01.md - m02-practice (Workspace) - Visual Studio Code". The status bar at the bottom indicates "Screen Reader Optimized", "Ln 7, Col 1", "Spaces: 4", "UTF-8", "CRLF", and "Markdown".

SS05-05. Two commits record distinct reviewed purposes.

As an additional staging check, edit a harmless sentence in the README, stage it, then edit that sentence again. Inspect both diff views without committing. Explain which version would enter a commit now. Restore your chosen final sentence in the editor and restage only if you intend to keep it. This shows that “staged file” does not mean every future edit to that file is automatically selected.

8. Failure and recovery: staged build output

Practice this only after the baseline commit exists. Create a harmless text file such as `build-note.tmp` in the practice root, deliberately stage it, and inspect the staged diff. It represents accidental generated output. Unstage this specific path while preserving the file:

```
# Context: PowerShell in the disposable practice repository, after a baseline commit.
git restore --staged -- build-note.tmp
git status
```

With `--staged`, this operation updates the staging area; it does not discard the working file. Add an appropriate exact ignore entry for the teaching temporary file, or use the project's existing `generated-output` directory in actual work. Then inspect `.gitignore` and stage its intended change. Avoid broad patterns that accidentally hide real fixtures or source. [git restore reference](#)

If generated output was already committed, an ignore entry alone will not untrack it. That is a different state requiring a deliberate tracked-file cleanup, outside this simple staging exercise. Identify the state first. Do not respond by deleting the project or using a destructive reset. The recovery here is complete when the unwanted path is absent from the staged snapshot and the useful work remains present.

Completion checklist

- ☐ I initialized the intended source-only folder and recorded local identity.
- ☐ I can distinguish working, staged, and committed contents.
- ☐ I inspected ordinary and cached diffs before each meaningful commit.
- ☐ I have two explanatory commits and relevant verification evidence.
- ☐ I recovered from staging harmless output without deleting useful files.

Review questions with answers

1. Does saving in VS Code create a commit? No. Saving updates the working file. Staging selects contents, and committing records the selected snapshot. These actions are separate even when an interface places their controls close together.

2. Why can one file appear staged and unstaged? You may have edited it after staging. The index contains one version and the working tree another. Compare both diffs to decide which content belongs in the next commit.

3. Does a clean status prove the tests pass? No. It describes repository state. A test result must come from a named command and source version. Keeping both observations gives a much more useful checkpoint.

Transfer exercise

Choose a recent change in a real project. Describe a commit boundary that would make its purpose easy to review and undo. Identify one file that belongs with the change and one generated or unrelated artifact that does not. Use that boundary during your next small task.

6. Put Your Project on GitHub

You will synchronize a practice project and reproduce it from a second checkout. Start with the two local commits from Chapter 5, a working toolchain, and a GitHub account you control. The repository can remain private. By the end you will have evidence that the intended remote contains your history and that another checkout can follow the written build path.

A successful local build answers an important question: the current source works with the current installed tools. It leaves another question open: did you preserve all the files and instructions required to repeat it? Pushing and cloning help expose missing tracked files and undocumented assumptions. The exercise is deliberately small so that a failure teaches you about reproduction rather than a sprawling deployment system.

1. Choose identity and visibility deliberately

Sign into GitHub in your browser and verify the account you intend to use for the practice repository. Do this before creating a repository or copying a remote URL. A browser can be signed into one account while Git's credential helper uses another. The account visible on a web page is useful information, but a successful Git operation is a separate authentication check.

Create a private practice repository with a neutral name such as `sensor-monitor-practice`. Private visibility satisfies this lesson; public sharing is not required. Inspect the selected owner and visibility before creation. The hosted service's Hello World guide explains repositories, branches, commits, and pull requests; our exercise adds reproduction checks to that introductory workflow. [GitHub Hello World](#)

Because you already have local history, create the remote without an initial README, license, or ignore file. The README and ignore file already exist in your CP01 project; licensing is a separate decision, and CP01 does not supply a license file. Creating any separate initial commit on the server introduces a second history to reconcile. If you accidentally initialize the remote, do not force-push over it; inspect both histories and choose a deliberate recovery.

Before the first push, inspect the actual tracked files and recent commits. A private repository still receives what you upload. The exercise includes source, scripts, fixtures, and documentation, not credentials or unrelated project history. Git can preserve deleted content in earlier commits, so reviewing only the current file tree is insufficient if you introduced a sensitive file earlier. Use an appropriate clean practice history from the supplied materials.

Figure SS06-01 shows the intended private repository before creation. The captured lesson uses book-only identity and content. Your account need not resemble that identity; it needs to be the one you knowingly selected. Do not reuse the author's remote URL as your exercise destination.

Create a new repository

Repositories contain a project's files and version history. Have a project elsewhere? [Import a repository](#).
Required fields are marked with an asterisk ().*

1 General

Owner * **Repository name ***

 aieembook-eng / sensor-monitor-course

✓ sensor-monitor-course is available.

Great repository names are short and memorable. How about [bookish-barnacle](#)?

Description

0 / 350 characters

2 Configuration

Choose visibility *
Choose who can see and commit to this repository

 **Private**

Add README
READMEs can be used as longer descriptions. [About READMEs](#)

Off ☐

Add .gitignore
.gitignore tells git which files not to track. [About ignoring files](#)

No .gitignore

SS06-01. The intended destination is private.

2. Connect the correct remote

A **remote** is a named repository location stored in local Git configuration. `origin` is a conventional name, not a guarantee that the address is right. Inspect it explicitly. From PowerShell in the Chapter 5 practice root:

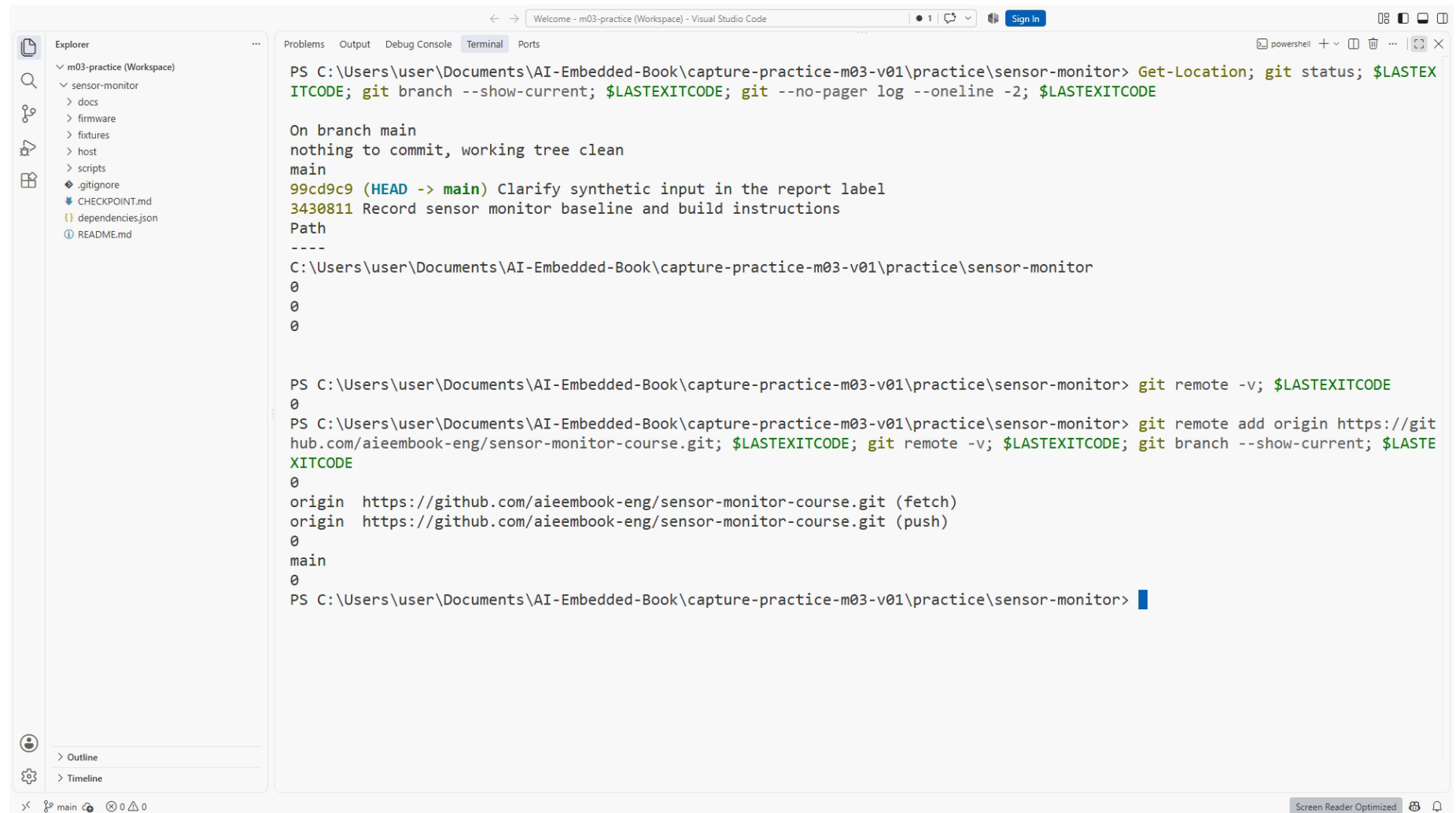
```
# Context: PowerShell in sensor-monitor-git, containing the two local commits.
git status
git branch --show-current
git remote -v
```

If no remote exists, copy the HTTPS URL from your newly created private repository and add it. Substitute your real account and repository name; the following address is a template, not an executable destination:

```
# Context: same repository; replace YOUR-ACCOUNT with your selected owner.
git remote add origin https://github.com/YOUR-ACCOUNT/sensor-monitor-practice.git
git remote -v
```

Read the result before pushing. Check owner, repository, and transport. If `origin` already exists, adding it again will fail rather than fix its value. Inspect the existing URL and, only after identifying it as wrong, set the correct URL with `git remote set-url origin` followed by the copied address. Record what you corrected. [git remote reference](#)

Do not put a token in a remote URL. A URL can appear in command output, shell history, logs, and screenshots. Use the supported credential flow instead. Figure SS06-02 shows the demonstration remote and branch after inspection, with no secret material. A correct-looking URL is a configuration observation; the next transport operation establishes whether access actually works.



The screenshot shows the Visual Studio Code interface with a terminal window open. The Explorer sidebar on the left shows the project structure for 'm03-practice (Workspace)', including 'sensor-monitor' and its subdirectories. The terminal window displays the following commands and output:

```
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor> Get-Location; git status; $LASTEXITCODE; git branch --show-current; $LASTEXITCODE; git --no-pager log --online -2; $LASTEXITCODE

On branch main
nothing to commit, working tree clean
main
99cd9c9 (HEAD -> main) Clarify synthetic input in the report label
3430811 Record sensor monitor baseline and build instructions
Path
----
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor
0
0
0

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor> git remote -v; $LASTEXITCODE
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor> git remote add origin https://github.com/aieembook-eng/sensor-monitor-course.git; $LASTEXITCODE; git remote -v; $LASTEXITCODE; git branch --show-current; $LASTEXITCODE
0
origin https://github.com/aieembook-eng/sensor-monitor-course.git (fetch)
origin https://github.com/aieembook-eng/sensor-monitor-course.git (push)
0
main
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor>
```

SS06-02. Inspect the actual origin and branch.

3. Authenticate without confusing it with commit identity

Your Chapter 5 `user.name` and `user.email` settings describe commit authorship. They do not grant access to a private repository. HTTPS Git authentication uses a credential mechanism supported by the host and installed client. GitHub documents Git Credential Manager as one supported way to manage HTTPS authentication. [Caching GitHub credentials](#)

When the client requests sign-in, complete the private browser flow and return to the terminal. Do not paste an account password or recovery code into a project file. The recorded lesson omits credential entry and shows the successful post-authentication operation. If a login challenge requires your action, complete it yourself; it is not a reason to manufacture a success result.

Authentication failure and repository-not-found messages need context. The URL might be wrong, the credential might belong to a different account, or the account might lack access. Verify the repository in the browser and configured remote first. Then use the supported credential-manager flow to correct the identity. Repeatedly retrying a push does not fix a wrong owner string.

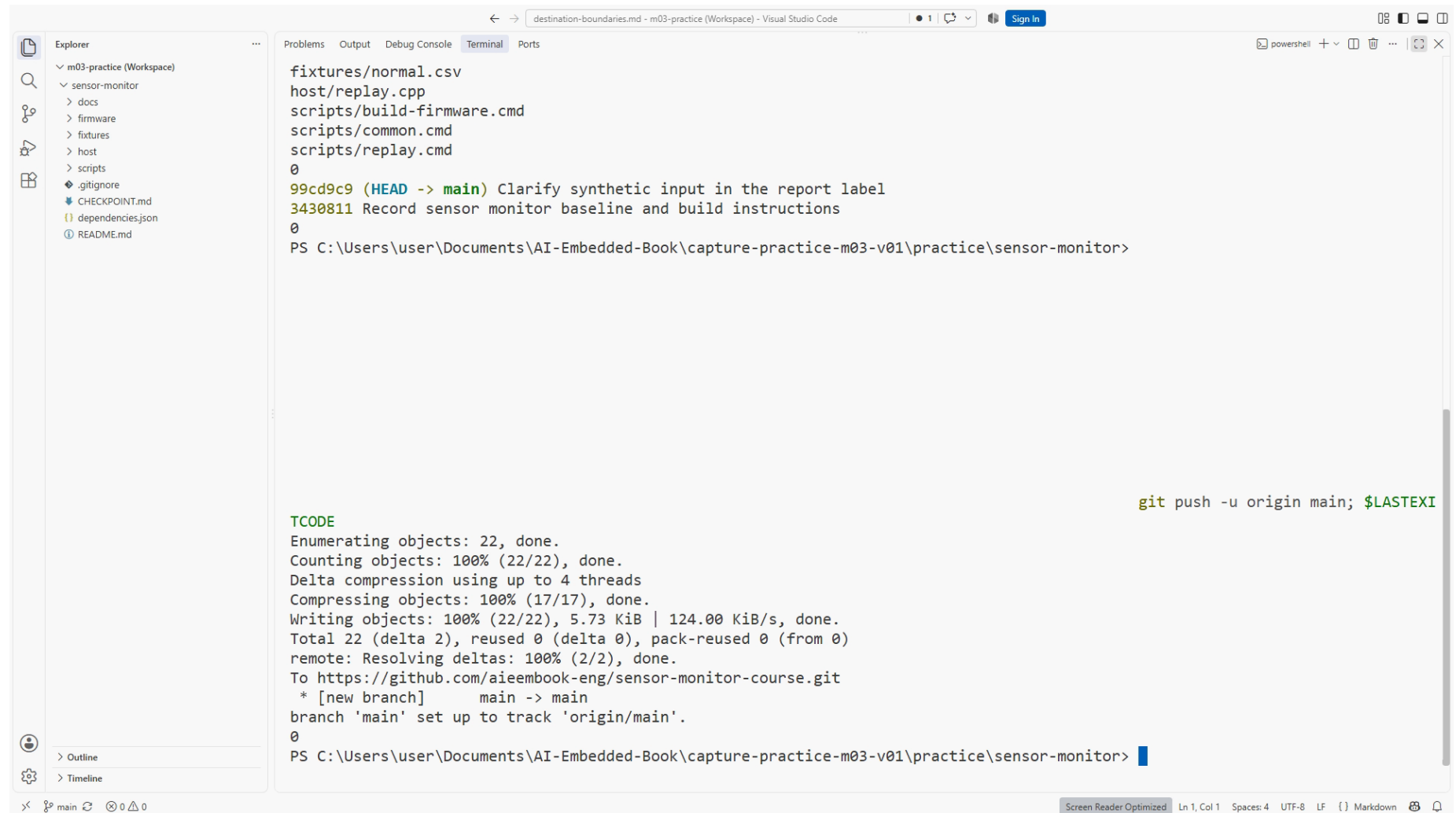
4. Push, fetch, and pull have different jobs

Push sends local commits and requests a remote reference update. **Fetch** retrieves remote objects and updates local knowledge of remote references without integrating them into your current working branch. **Pull** combines fetching with integration according to its options and configuration. In this lesson, `--ff-only` allows a pull only when the branch can advance without creating a merge commit. [git push](#), [git fetch](#), [git pull](#)

From the correct practice repository, after reviewing its clean state:

```
# Context: PowerShell in the reviewed local practice repository.
git push -u origin main
git fetch origin
git status
git log --oneline --decorate -3
```

The `-u` option records an upstream relationship for the pushed branch. Inspect the actual output for success and the named remote branch. A nonzero error is not a partial success to ignore. Figure SS06-03 records the initial push and tracking result; SS06-04 shows the expected files and latest teaching commit on GitHub.



The screenshot shows the Visual Studio Code interface with a terminal window open. The Explorer sidebar on the left shows a workspace named 'm03-practice' containing a 'sensor-monitor' folder with subfolders 'docs', 'firmware', 'fixtures', 'host', and 'scripts'. It also contains files like '.gitignore', 'CHECKPOINT.md', 'dependencies.json', and 'README.md'. The terminal window is titled 'destination-boundaries.md - m03-practice (Workspace) - Visual Studio Code' and shows the following content:

```
fixtures/normal.csv
host/replay.cpp
scripts/build-firmware.cmd
scripts/common.cmd
scripts/replay.cmd
0
99cd9c9 (HEAD -> main) Clarify synthetic input in the report label
3430811 Record sensor monitor baseline and build instructions
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor>

git push -u origin main; $LASTEXI

TCODE
Enumerating objects: 22, done.
Counting objects: 100% (22/22), done.
Delta compression using up to 4 threads
Compressing objects: 100% (17/17), done.
Writing objects: 100% (22/22), 5.73 KiB | 124.00 KiB/s, done.
Total 22 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/aieembook-eng/sensor-monitor-course.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor>
```

The status bar at the bottom indicates 'Screen Reader Optimized', 'Ln 1, Col 1', 'Spaces: 4', 'UTF-8', 'LF', and 'Markdown'.

SS06-03. The initial history is pushed successfully and the branch tracks its upstream.

The screenshot shows a GitHub repository page for 'sensor-monitor-course' by 'aieembook-eng'. The repository is private and has 0 Watchers, 0 Forks, and 0 Stars. The main branch is 'main' with 1 Branch and 0 Tags. The repository contains a commit titled 'Clarify synthetic input in the report label' by 'aieembook-eng' 7 hours ago, with 2 Commits. The file list includes:

File	Description	Time
docs	Record sensor monitor baseline and build instructions	8 hours ago
firmware/sensor-monitor	Record sensor monitor baseline and build instructions	8 hours ago
fixtures	Record sensor monitor baseline and build instructions	8 hours ago
host	Record sensor monitor baseline and build instructions	8 hours ago
scripts	Record sensor monitor baseline and build instructions	8 hours ago
.gitignore	Record sensor monitor baseline and build instructions	8 hours ago
CHECKPOINT.md	Record sensor monitor baseline and build instructions	8 hours ago
README.md	Clarify synthetic input in the report label	7 hours ago
dependencies.json	Record sensor monitor baseline and build instructions	8 hours ago

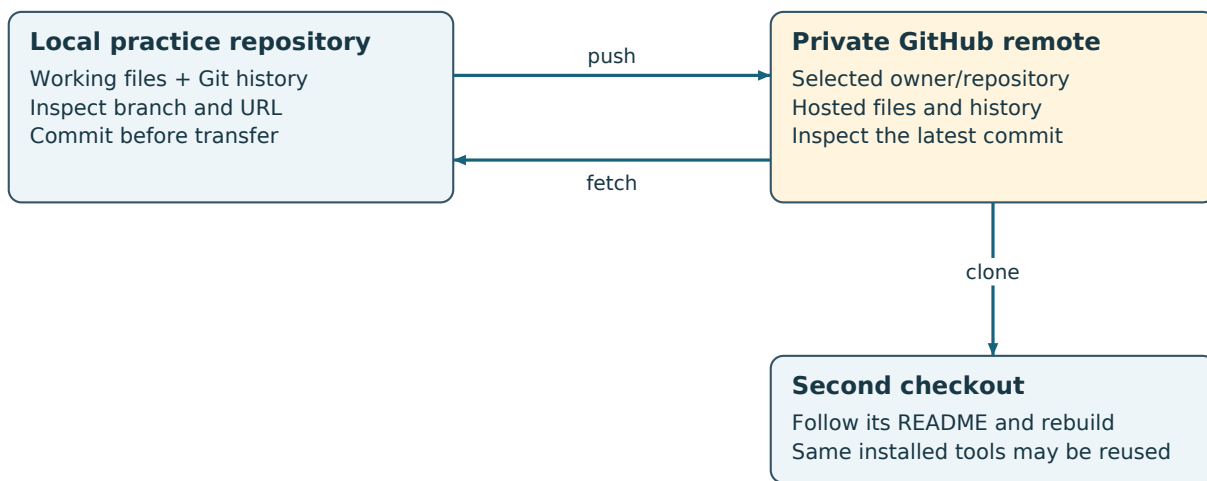
The right sidebar shows the 'About' section with no description, website, or topics provided. It also lists 'Releases' (No releases published, [Create a new release](#)) and 'Packages' (No packages published, [Publish your first package](#)). The 'Contributors' section shows 1 contributor: 'aieembook-eng'.

SS06-04. The hosted repository shows the expected files and latest demonstration commit.

Open the repository in the browser and inspect the README, source, and history. Compare the latest commit identifier with your local `git rev-parse HEAD` result. This cross-check catches an easy mistake: looking at the wrong repository tab while assuming a successful push updated it. It also helps distinguish a stale page from a transport problem.

Figure D06 shows a local repository and a remote with separate arrows for push, fetch, and clone. Fetching is valuable even when you do not immediately integrate changes, because it lets you examine what exists before choosing an action. We use that habit when recovering from a remote-ahead state.

D06 · Local history and a verified remote



Fetch does not integrate into the working branch. Pull includes an integration step.

D06. Push sends commits to the remote; fetch retrieves remote knowledge without integration; clone creates a separate checkout for reproduction.

5. Worked example: reproduce from another folder

Close terminals that might tempt you to keep using the first checkout. Open PowerShell in the parent learning folder, such as `C:\BookLab`. Clone the private repository to a deliberately new name:

```
# Context: PowerShell in C:\BookLab, outside the original repository.
git clone https://github.com/YOUR-ACCOUNT/sensor-monitor-practice.git sensor-monitor-repro
Set-Location .\sensor-monitor-repro
Get-Location
git status
git rev-parse HEAD
```

Replace the URL with the already verified practice URL. Git creates the destination folder and checks out the default branch. If the destination exists and contains work, choose a new empty

path rather than delete it. Figure SS06-05 shows the new checkout path and completed clone.
[git clone reference](#)

The screenshot shows a Visual Studio Code interface with a terminal window open. The terminal displays the output of several git commands executed in a PowerShell prompt. The commands include fetching the origin, checking status, and cloning a repository into a new checkout. The output shows successful completion of these operations, including object compression and resolution of deltas.

```

Delta compression using up to 4 threads
Compressing objects: 100% (17/17), done.
Writing objects: 100% (22/22), 5.73 KiB | 124.00 KiB/s, done.
Total 22 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/aieembook-eng/sensor-monitor-course.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor> git fetch origin; $LASTEXITCODE;
git status -sb; $LASTEXITCODE; git rev-parse HEAD; $LASTEXITCODE; git rev-parse origin/main; $LASTEXITCODE
0
## main...origin/main
0
99cd9c9a689888ccf46893e4621b2eaa6f1b045d
0
99cd9c9a689888ccf46893e4621b2eaa6f1b045d
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor> Get-Location; git clone -c core.a
utocrlf=false -c core.eol=lf -- https://github.com/aieembook-eng/sensor-monitor-course.git 'C:\Users\user\Documents\AI-Embedded
-Book\capture-practice-m03-v01\sensor-monitor-repro'; $LASTEXITCODE

Cloning into 'C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\sensor-monitor-repro'...
remote: Enumerating objects: 22, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 22 (delta 2), reused 22 (delta 2), pack-reused 0 (from 0)
Receiving objects: 100% (22/22), done.
Resolving deltas: 100% (2/2), done.
Path
----
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor
0

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor>

```

SS06-05. Clone the private repository into a separate checkout.

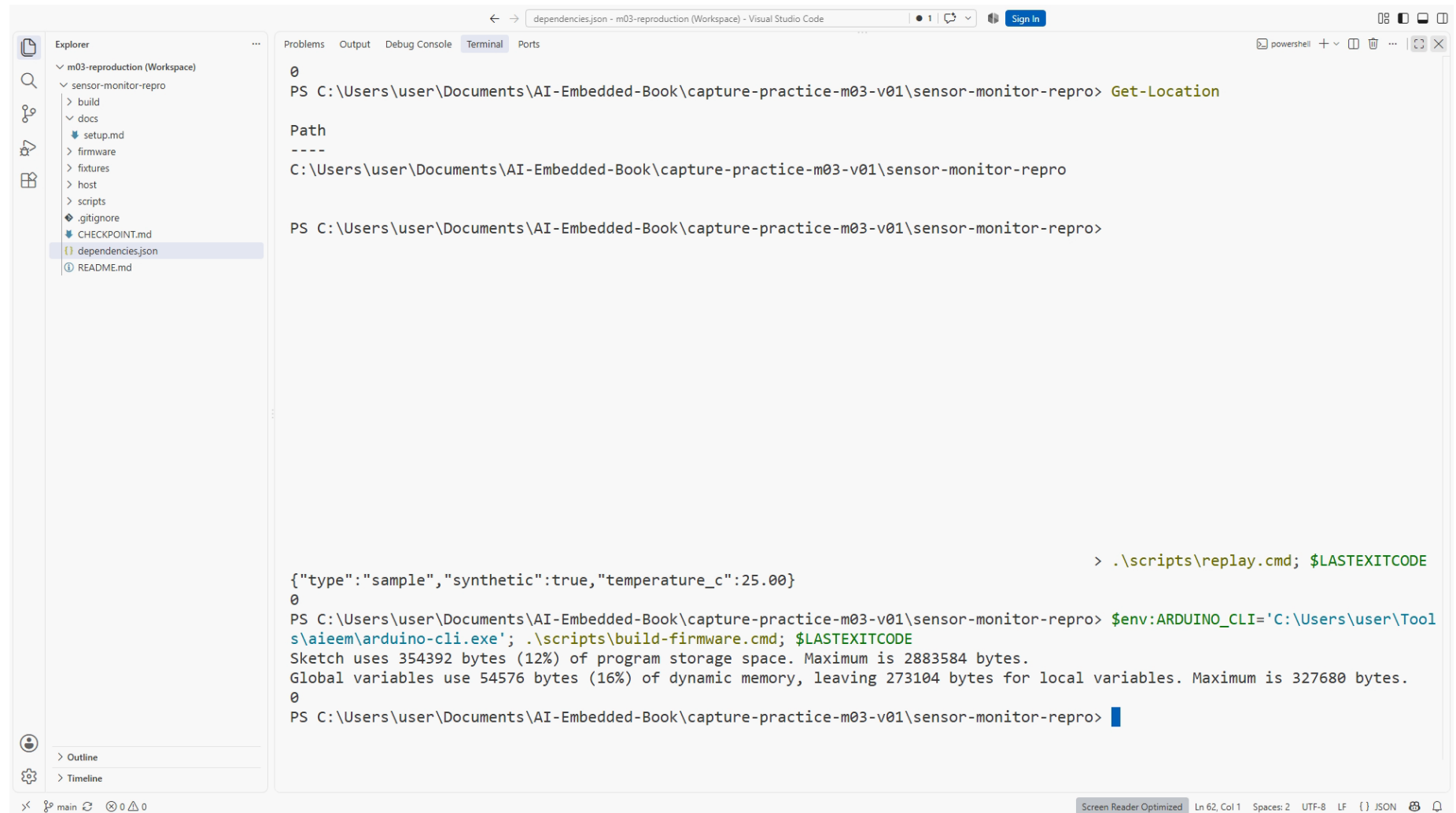
Follow only the new checkout's README and dependency record. Run the baseline checks from that root:

```
# Context: PowerShell in the newly cloned sensor-monitor-repro root.  
.\scripts\replay.cmd  
.\scripts\build-firmware.cmd
```

Run each command separately, capture the result, and label the replay synthetic. The same installed toolchain may be reused; a second checkout is not a fresh operating-system installation. This check tests whether tracked files and written instructions are sufficient in the recorded environment. It does not prove the setup works on every machine or from an empty dependency cache.

A useful reproduction record includes date, cloned URL without credentials, commit, destination, tool versions, commands, exit results, and limitations. If a build succeeds only after copying an untracked header from the original folder, the reproduction test has found an omission. Add the necessary source to the intended project, explain it, and repeat the affected clone/build check after synchronizing.

Figure SS06-06 shows the real second-checkout build when captured. The authentication and clone checks in the edition's earlier readiness rehearsal established transport for a bounded setup repository; they were not the complete project reproduction exercise. Keep those scopes distinct when reporting your own result.



The screenshot shows the Visual Studio Code interface with a workspace named 'm03-reproduction'. The Explorer sidebar on the left shows a folder structure for 'sensor-monitor-repro' containing subfolders like 'build', 'docs', 'firmware', 'fixtures', 'host', 'scripts', and files like '.gitignore', 'CHECKPOINT.md', 'dependencies.json', and 'README.md'. The 'dependencies.json' file is selected. The Terminal window at the bottom shows a PowerShell session with the following commands and output:

```
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\sensor-monitor-repro> Get-Location

Path
----
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\sensor-monitor-repro

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\sensor-monitor-repro>

> .\scripts\replay.cmd; $LASTEXITCODE

{"type":"sample","synthetic":true,"temperature_c":25.00}
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\sensor-monitor-repro> $env:ARDUINO_CLI='C:\Users\user\Tools\aiem\arduino-cli.exe'; .\scripts\build-firmware.cmd; $LASTEXITCODE
Sketch uses 354392 bytes (12%) of program storage space. Maximum is 2883584 bytes.
Global variables use 54576 bytes (16%) of dynamic memory, leaving 273104 bytes for local variables. Maximum is 327680 bytes.
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\sensor-monitor-repro>
```

SS06-06. The separate checkout builds successfully using the recorded setup.

6. Lab LAB06: synchronize and rebuild independently

Create or use your private practice repository, verify its remote, push the two meaningful commits, and clone into a new folder. Before building, write down which checkout you are in and its commit. Follow the README without referring to unsaved notes in the original editor session. Save actual CP01 fixed-sample replay and cross-build results. Automated assertions are introduced at CP07; do not invent them at this checkpoint.

The expected observations are matching intended history, a clean initial second checkout, and successful documented checks. Hashes, elapsed time, paths, and compiled size can differ from another learner's run because environment and metadata differ. Explain meaningful differences rather than demanding that every character match a screenshot.

If the README omits a prerequisite, record the failure and improve the instruction in a small new commit. Then synchronize and retry the affected check from a clean source state. A reproduction test that reveals a missing step is doing useful work. The lab is complete when you can follow the corrected path and explain its evidence limits.

For an extra inspection, compare `git ls-files` in the original and cloned folders. This lists tracked paths, which helps distinguish source omitted from history from a toolchain dependency installed outside it. Do not solve a missing dependency by committing arbitrary installed programs. Record how the supported dependency is obtained and which version was tested.

7. Failure and recovery: the remote is ahead

Git may reject a push when the requested update would discard remote work. First preserve local changes and inspect state. In this beginner workflow, start from a clean working tree, then fetch and compare:

```
# Context: PowerShell in the local practice repository; inspect before integrating.
git status
git fetch origin
git log --oneline --graph --decorate --all -8
```

If your branch has no unique commits and can simply advance, use `git pull --ff-only`. If local and remote branches both contain unique commits, the command should stop rather than invent an integration choice for you. Read the changed files and use the review/merge lesson next. GitHub's non-fast-forward guidance explains why preserving remote history matters.

Non-fast-forward errors

Do not use force-push as a generic repair. It changes the question from “how do I incorporate existing work?” to “may I replace the remote reference despite its current history?” That is not this lesson's objective. If you created the remote with an unrelated initial commit, one option for a disposable exercise is another empty private practice repository with a new name, preserving the first until you resolve what it contains.

For a wrong remote, inspect and correct the URL before any further push. For authentication failure, correct the private credential flow. For a build failure after clone, compare tracked source and documented dependencies. These failures may appear during the same session, but they require different remedies.

8. Why a pull request exists

A **pull request** is a place to propose and discuss integrating changes between branches. It gathers a description, diff, comments, and related checks. It is not proof that code is good, and opening one does not require public visibility. In the next chapter you will use a branch and review a bounded agent change before integration.

Write a useful description around behavior: “The README now says synthetic input explicitly. The fixed sample and firmware are unchanged. Reviewed the wording and ran the applicable CP01 replay.” Include only results that occurred. If checks are pending, say so. A reviewer should be able to decide what to inspect without reading your entire conversation with an assistant.

Completion checklist

- ☐ I chose the intended account and private visibility deliberately.
- ☐ I inspected the remote before pushing and verified hosted history afterward.
- ☐ I cloned into a distinct folder and recorded the actual commit and location.
- ☐ I reproduced the documented checks and stated their environment limits.
- ☐ I can distinguish authentication, history, path, and build failures.

Review questions with answers

1. Does fetch change my working source? Fetch updates local knowledge of the remote without integrating its changes into the current working branch. Inspect that knowledge before choosing a merge or fast-forward action.

2. Why build from a second checkout on the same computer? It can reveal missing tracked files and dependence on the first folder's leftovers. It does not prove a completely fresh machine setup, so record that limitation.

3. Why avoid force-push when the remote is ahead? The remote may contain work you have not incorporated. Fetching and reviewing preserves your ability to integrate it deliberately instead of replacing its reference blindly.

Transfer exercise

Choose a project you could keep private. Write a three-step reproduction check naming the commit, dependency record, and success evidence. Identify one assumption a same-machine clone would leave untested. That distinction will make your future release notes more useful.

7. Experiment, Review, and Recover

You will use a branch for a bounded task, reject an unrelated edit, resolve a small conflict, and undo a committed fault without rewriting shared history. Start with Chapter 6 and a separate copy of the supplied Git practice resources. Keep your useful project intact. Recovery commands affect different kinds of data, so we name the starting state before every example.

The value of an experiment depends partly on your ability to judge and reverse it. An assistant's patch can contain a correct reporting change and an unrelated host sample-value change. Accepting the whole patch because one part looks good confuses those decisions. Git gives you ways to isolate work and inspect exactly what is proposed; your brief supplies the criteria for accepting it.

1. Start one branch for one task

A **branch** is a named line of development whose reference advances as you commit. A new branch lets you develop a change while keeping a clear reference to the previous line. It is not a second physical copy of every file, and it does not automatically isolate uncommitted working changes. Begin this exercise from a clean, known baseline. [Git branching basics](#)

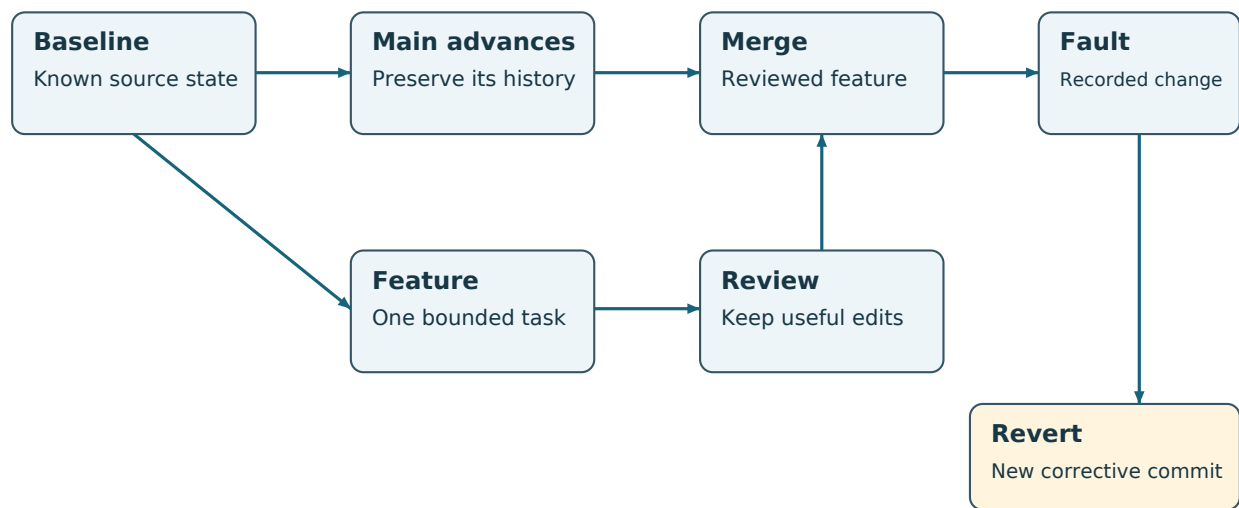
In PowerShell, inside the disposable practice repository:

```
# Context: PowerShell in the Git practice copy, starting with clean main.
git status
git branch --show-current
git log --oneline -3
git switch -c feature/report-origin
git branch --show-current
```

The expected branch is now `feature/report-origin`. If Git says the name already exists, inspect it instead of creating a similarly named branch by guesswork. If status was dirty, preserve and account for those changes first. Creating a branch does not make unexplained edits disappear.

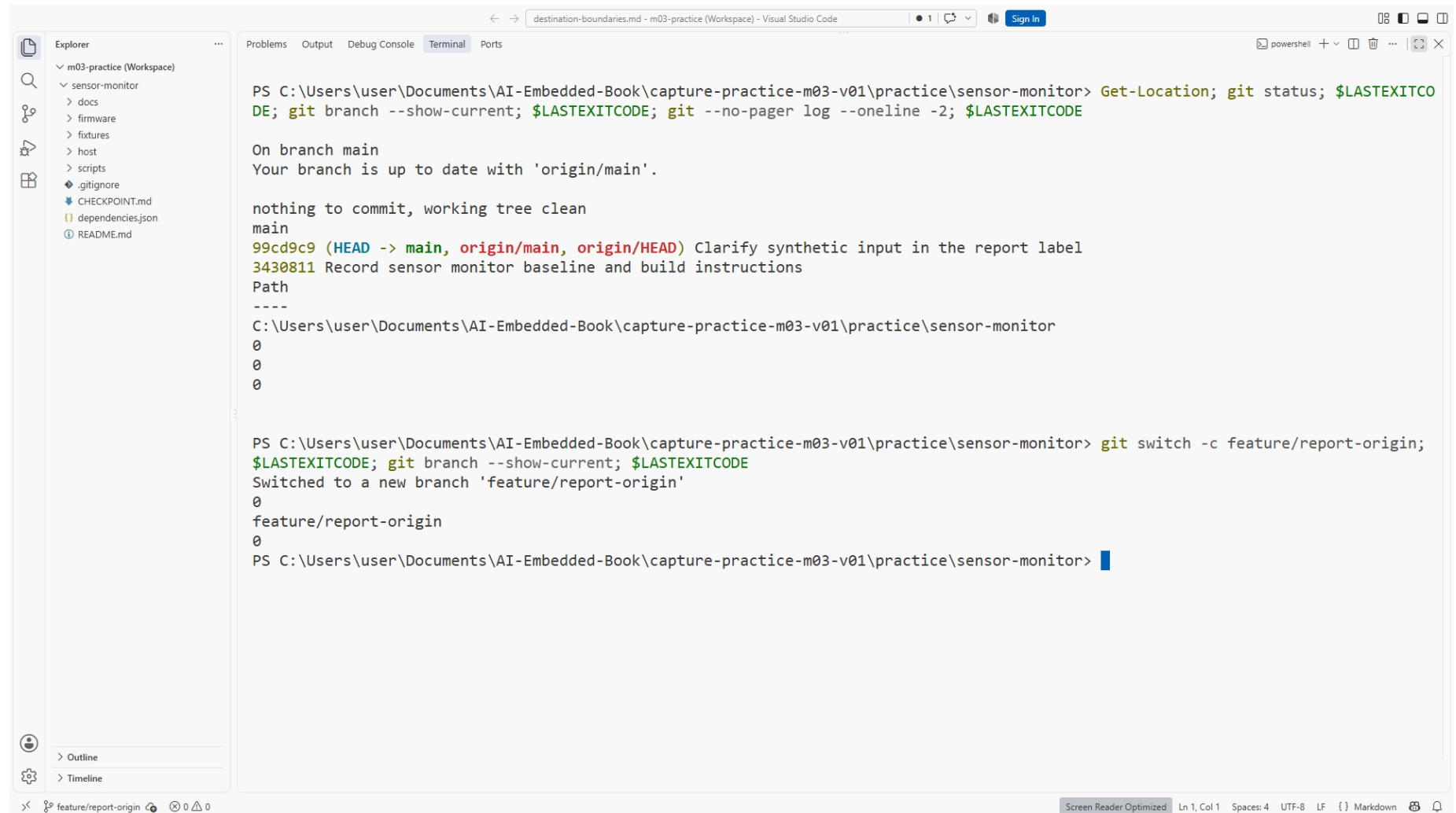
Figure SS07-01 shows the actual baseline and selected branch. Figure D07 shows the later path through review, merge, and a subsequent revert. A revert is a new event in history, so the diagram retains the original mistaken commit rather than erasing it.

D07 · Review, merge, and record a reversal



Working-tree restore is a different operation. A revert preserves the earlier fault in history.

D07. A feature develops from a baseline, is reviewed and merged, and a later fault can be reversed by a new commit while the faulty commit remains in history.



The screenshot shows a Visual Studio Code interface with a terminal window open. The terminal is running PowerShell commands to check the current Git branch and create a new feature branch. The Explorer sidebar on the left shows the project structure for 'm03-practice (Workspace)', including folders like 'docs', 'firmware', 'fixtures', 'host', 'scripts', and files like '.gitignore', 'CHECKPOINT.md', 'dependencies.json', and 'README.md'. The terminal output shows the current branch is 'main' and it is up to date with 'origin/main'. The user then switches to a new branch named 'feature/report-origin'.

```
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor> Get-Location; git status; $LASTEXITCODE; git branch --show-current; $LASTEXITCODE; git --no-pager log --oneline -2; $LASTEXITCODE

On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
main
99cd9c9 (HEAD -> main, origin/main, origin/HEAD) Clarify synthetic input in the report label
3430811 Record sensor monitor baseline and build instructions
Path
----
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor
0
0
0

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor> git switch -c feature/report-origin; $LASTEXITCODE; git branch --show-current; $LASTEXITCODE
Switched to a new branch 'feature/report-origin'
0
feature/report-origin
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor>
```

SS07-01. Create the feature branch from the correct clean baseline.

2. Ask for a bounded agent change

Use the Chapter 3 pattern. A suitable task is: “Add a clear synthetic-origin statement to the report documentation. Change only the designated README paragraph. Preserve the fixed host sample, firmware, dependencies, and build commands. Show the diff and distinguish checks actually run from proposed checks.” Give the agent the repository root and relevant requirement.

Watch the action record and inspect the resulting files. If the actual agent produces only the requested change, do not pretend it also made the unrelated edit in this lesson. Use the supplied **seeded review candidate** for the teaching comparison. It deliberately includes an extra fixed-sample edit in `host/replay.cpp`. Label it as a constructed fault exercise, not as evidence of an agent failure that did not occur.

The **seeded review candidate** contains two ideas: an intended documentation sentence saying readings are synthetic, and an unrelated change of the fixed host sample from 25.00 to 30.00. CP01 has no threshold policy yet. Both are visible in the diff. The first serves the brief; the second changes behavior without support from the task. The candidate can be partly useful without being acceptable as a whole.

Figure SS07-02 must show the candidate under discussion and identify whether it came from a live agent turn or supplied fault setup. This provenance matters because the reader is learning to inspect evidence, not watching an invented story about an assistant.

```

CISE: host change seeded by instructor, not the agent'; git diff --stat; $LASTEXITCODE; git --no-pager diff --; $LASTEXITCODE
CONSTRUCTED REVIEW EXERCISE: host change seeded by instructor, not the agent
  README.md      | 5 +++--
  host/replay.cpp | 2 +-
  2 files changed, 4 insertions(+), 3 deletions(-)
0
diff --git a/README.md b/README.md
index c185fc4..faefcf1 100644
--- a/README.md
+++ b/README.md
@@ -1,7 +1,8 @@
 # Sensor Monitor - CP01

-Synthetic source-reading baseline for Chapters 5-12. It prints one fixed JSON
-reading on the host, and repeats that reading in firmware. It does not yet have
+Synthetic source-reading baseline for Chapters 5-12. The host demonstration prints
+one fixed synthetic JSON sample; it does not measure a sensor. Firmware repeats
+that reading. It does not yet have
  fault policy, a fixture parser, native assertions, commands or alert behavior.
  Report label: synthetic input.

diff --git a/host/replay.cpp b/host/replay.cpp
index 82eb724..0b59008 100644
--- a/host/replay.cpp
+++ b/host/replay.cpp
@@ -1,6 +1,6 @@
 #include <iostream>
 // A minimal source-reading baseline. The full replay adapter arrives in CP05.
 int main() {
-  std::cout << "{\"type\":\"sample\", \"synthetic\":true, \"temperature_c\":25.00}\n";
+  std::cout << "{\"type\":\"sample\", \"synthetic\":true, \"temperature_c\":30.00}\n";
   return 0;
 }
0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\practice\sensor-monitor>

```

SS07-02. Review the intended edit and the unrelated constructed edit.

3. Review the diff against the requirement

Inspect the changed-file list first, then each hunk. A **hunk** is a grouped section of changes with context. Ask what behavior each hunk changes and which criterion justifies it. A README-only task touching policy source needs explanation before acceptance. Small does not mean harmless: a one-character comparison can change a boundary case.

```
# Context: PowerShell on feature/report-origin after the candidate edit.
git status
git diff --stat
git diff
```

Write a review note with finding, evidence, consequence, and requested correction. For example: “Reject the 25.00-to-30.00 sample change in `host/replay.cpp`. The task permits a README clarification and preserves the fixed sample. Restore that hunk while keeping the README improvement, then run the CP01 replay.” This is more actionable than “please improve code quality.”

Use the editor to remove only the unrelated candidate change, or apply the supplied correction in the disposable exercise as documented. Inspect the diff again. If a file contains useful and unwanted edits, replacing the whole file from a baseline could discard useful work; inspect individual hunks before choosing that route. Save and review the final candidate before staging.

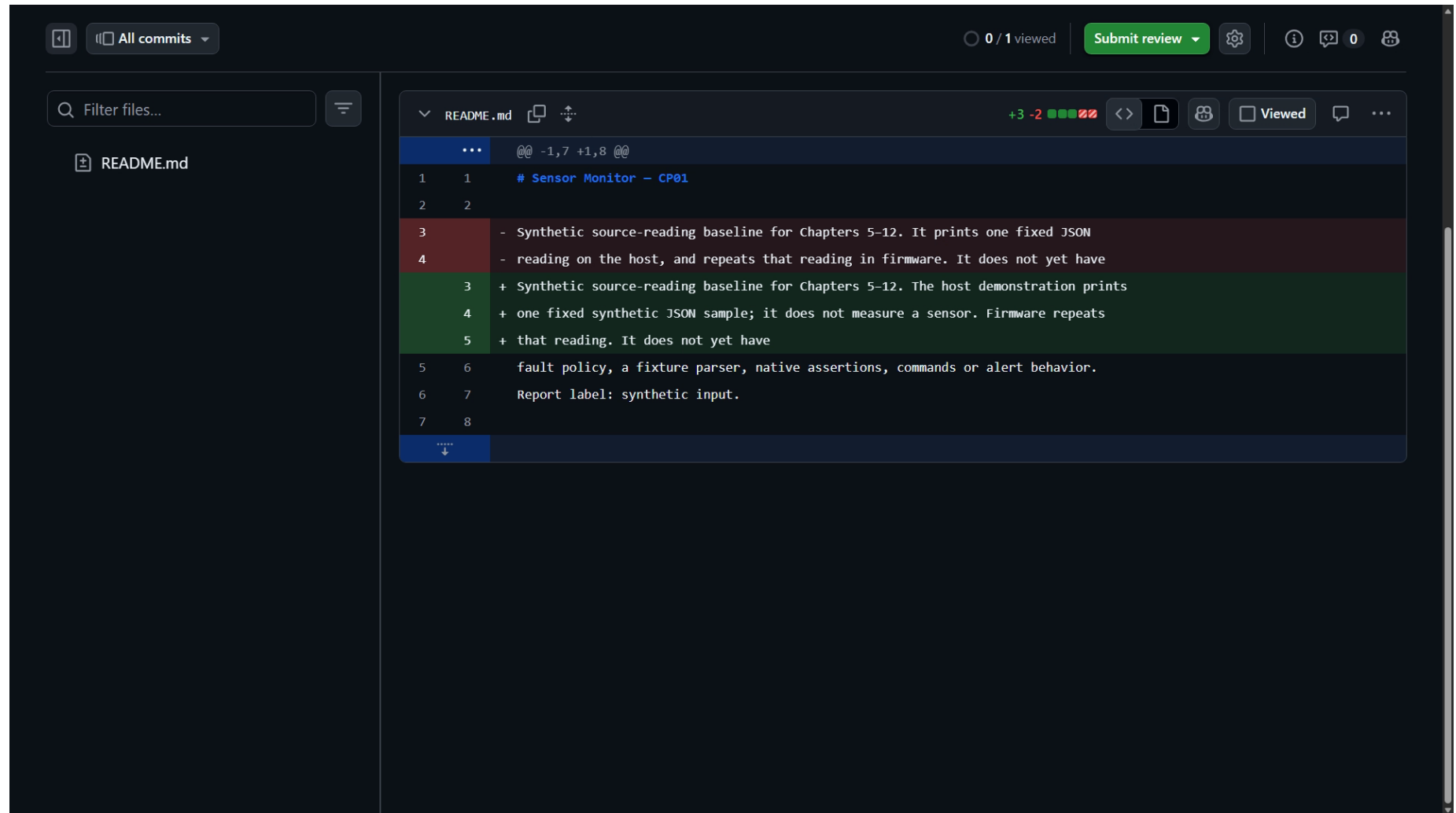
Run checks proportionate to the result. For a pure explanatory sentence, inspect its accuracy and links. If the host source changed, run the CP01 replay after correction. Its expected fixed JSON sample is documented in README; automated assertions arrive later. Record verification you did not perform. A reviewer should not have to infer test coverage from the word “checked.”

4. Propose a pull request

Commit the reviewed task, push the feature branch to your private practice remote, and open a pull request targeting `main`. Inspect the base and compare branches before creation. In Files changed, verify that the remote proposal matches the local reviewed diff. GitHub documents reviewing changed files and commenting on a pull request. [Reviewing proposed changes](#)

A concise description names the problem, new behavior, and actual validation. For this exercise: “The README now explicitly identifies replay inputs as synthetic. Reviewed the wording against the evidence requirement; no physical result is claimed.” Add replay details only if it ran. If this is a solo exercise, use a written self-review; do not claim another person reviewed it.

Figure SS07-03 shows the real private pull-request diff. You can practice the local review while account access is unavailable, but keep the remote exercise pending until you perform it. A local diff is not evidence that a pull request was opened or merged.



SS07-03. The private pull request contains only the reviewed README clarification.

When the proposal is ready, integrate it using the practice repository's chosen merge method and update your local branch through the documented synchronization path. Preserve branch and commit references in your notebook. We do not need to rewrite earlier commits to produce a readable learning history.

5. Worked example: create and resolve a text conflict

Use a **fresh disposable clone of the supplied history bundle**. It contains generic manufactured teaching history, not authoring history. The two prepared conflict branches both change the CP01 README line. Work from the folder holding your downloaded `sensor-monitor-history.bundle`:

```
# Context: PowerShell in the learner download folder; choose a new destination.
git clone .\sensor-monitor-history.bundle chapter-07-conflict
Set-Location .\chapter-07-conflict
git switch -c resolve-origin origin/practice-conflict-left
git status
git merge origin/practice-conflict-right
git status
```

The left branch says Report label: teaching input. and the right says Report label: generated input. Neither is a physical observation. Read both against the requirement to preserve explicit synthetic origin.

The expected result is a conflict in README. Exact output wording varies. Open the file; Git's conflict markers separate the competing text. The upper portion is the current branch's contribution in this ordinary merge, and the lower portion is the branch being merged. Read the labels and context rather than treating “current” as a synonym for “correct.” [git merge reference](#)

The intended resolution is a deliberate third sentence: Report label: synthetic teaching input. It preserves the useful intent of both changes. Remove all conflict markers and rejected text. Preview the README to ensure it reads normally. Then stage and complete the merge:

```
# Context: same repository, after editing the complete resolution.
git add README.md
git diff --cached
git status
git commit -m 'Resolve report origin wording'
git status
```

Staging marks the path resolved; it does not judge the prose. You supply that judgment. Search the resolved file for leftover marker lines and read the complete paragraph. Figures SS07-04 and SS07-05 show the real competing text and verified resolution. Clean status after the merge is one observation; correct resolved meaning is another.

Visual Studio Code interface showing a merge conflict in the file `README.md`. The Explorer sidebar on the left shows the workspace structure: `m03-conflict (Workspace)` containing `docs`, `firmware`, `fixtures`, `host`, `scripts`, `.gitignore`, `CHECKPOINT.md`, `dependencies.json`, and `README.md` (which has a conflict icon). The main editor window displays the content of `README.md` with a conflict resolution overlay. The text in the editor is as follows:

```

1  # Sensor Monitor – CP01
2
3  Synthetic source-reading baseline for Chapters 5–12. It prints one fixed JSON
4  reading on the host, and repeats that reading in firmware. It does not yet have
5  fault policy, a fixture parser, native assertions, commands or alert behavior.
6  <<<<<< HEAD (Current Change)
7  Report label: teaching input.
8  =====
9  Report label: generated input.
10 >>>>>> origin/practice-conflict-right (Incoming Change)
11
12 From this folder in PowerShell:
13
14 ```powershell
15 .\scripts\replay.cmd
16 .\scripts\build-firmware.cmd
17 ```
18
19 Expected host line: `{"type":"sample","synthetic":true,"temperature_c":25.00}`.
20 Read [setup](docs/setup.md) and [dependencies](dependencies.json). No board is
21 needed. Compilation does not establish physical operation. Recover by extracting
22 this checkpoint ZIP to a new exercise folder. Chapter 5 initializes its own Git
23 repository; the separate history bundle provides prepared branch/revert examples.
24
25 Review each diff before saving a commit.
26

```

The conflict resolution overlay shows options: `Accept Current Change`, `Accept Incoming Change`, `Accept Both Changes`, and `Compare Changes`. The `<<<<<< HEAD (Current Change)` section is highlighted in green, and the `>>>>>> origin/practice-conflict-right (Incoming Change)` section is highlighted in blue. The bottom status bar indicates `resolve-origin!` and `0` conflicts. The bottom right corner shows `Screen Reader Optimized`, `Ln 6, Col 1`, `Spaces: 4`, `UTF-8`, `LF`, and `Markdown`.

SS07-04. Show both competing text changes in the genuine merge conflict.

```

0
On branch resolve-origin
Your branch is up to date with 'origin/practice-conflict-left'.

All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:
  modified:   README.md

0
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\conflict> git commit -m 'Resolve report origin wording'; $LAST
EXITCODE; git status; $LASTEXITCODE; git --no-pager log --oneline --graph -4; $LASTEXITCODE; Get-Content -Encoding UTF8 README.md |
Select-Object -First 6
[resolve-origin d2143e5] Resolve report origin wording
0
On branch resolve-origin
Your branch is ahead of 'origin/practice-conflict-left' by 2 commits.
(use "git push" to publish your local commits)

nothing to commit, working tree clean
0
*   d2143e5 (HEAD -> resolve-origin) Resolve report origin wording
| \
| * a5ac88b (origin/practice-conflict-right) Revise the reporting label for conflict practice
| * a22f8c6 (origin/practice-conflict-left) Revise the reporting label for conflict practice
| /
* 1ca7e71 (tag: CP01) CP01: record the chapter starting state
0
# Sensor Monitor - CP01

Synthetic source-reading baseline for Chapters 5-12. It prints one fixed JSON
reading on the host, and repeats that reading in firmware. It does not yet have
fault policy, a fixture parser, native assertions, commands or alert behavior.
Report label: synthetic teaching input.
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\conflict>

```

SS07-05. Resolve the conflict, verify the intended content, and confirm a clean state.

If you are not ready to resolve, use `git merge --abort` in this clean-start exercise and inspect status. The starting state matters because aborting a merge may not reconstruct unrelated edits that existed beforehand. Preserve work before entering a complex operation rather than expecting a recovery command to guess your intentions.

6. Restore an uncommitted edit deliberately

In a separate disposable restore example with an existing clean commit, change a harmless README sentence and do not stage it. Inspect `git diff -- README.md`. If you decide to discard that edit completely, run:

```
# Context: disposable restore example; only the unwanted README edit is unstaged.
git restore -- README.md
git diff -- README.md
git status
```

By default this restores the working file from the index. Because this example began clean and had no staged change, that index matches the baseline commit. The unwanted uncommitted edit is discarded. This differs from `git restore --staged`, which changes staging selection while preserving the working file. [git restore reference](#)

Before restoring, copy out text you may want to keep or commit useful work on an appropriate branch. Do not use a whole-directory restore when you intend to discard one known file edit. If you staged a useful version and then edited again, understand both snapshots before restoring either. The command is precise when the starting state is precise.

7. Revert a committed fault

Use another clone of the supplied bundle and create a branch from `FAULT-hysteresis`. This prepared later-project checkpoint includes the tests needed for a historical recovery exercise. You do not need to implement its capstone policy now. Its isolated bad commit uses `<` where the requirement calls for `<=` at the clear boundary of 27.0 °C. This differs from the introductory 28.0 °C warning-entry example.

```
# Context: PowerShell in the download folder; use a separate new destination.
git clone .\sensor-monitor-history.bundle chapter-07-revert
Set-Location .\chapter-07-revert
git switch -c recover-hysteresis FAULT-hysteresis
git status
git log --oneline -4
git show --stat HEAD
git show HEAD
.\scripts\test.cmd
```

Save the actual failing boundary check before recovery. If compilation or executable startup fails instead, diagnose that separately; it is not a reproduced policy failure. No physical alarm failure is implied.

Identify the exact seeded-fault commit from the log. In this prepared exercise it is HEAD, the current commit. If yours differs, stop and identify it rather than copy a hash from the book. Then create a revert:

```
# Context: clean fault exercise; HEAD has been verified as the isolated bad commit.
git revert --no-edit HEAD
.\scripts\test.cmd
.\scripts\build-firmware.cmd
git log --oneline -3
git status
```

git revert creates a new commit reversing the selected change. The bad commit remains visible, and the new commit explains its reversal. That is useful for shared history because others can follow what happened without earlier references being silently replaced. Conflicts can still occur when intervening changes overlap; this beginner exercise uses an isolated fault to expose the basic operation. [git revert reference](#)

Expected observations are a failing pre-revert test, a new revert commit, and a passing post-revert test for the same fixture. Save actual results. Figure SS07-06 shows those results and history. Reverting does not prove universal correctness; it removes that change and lets the relevant regression check establish restored behavior.

```

--- a/firmware/sensor-monitor/monitor_policy.cpp
+++ b/firmware/sensor-monitor/monitor_policy.cpp
@@ -25,7 +25,7 @@ void Monitor::sample_at(uint32_t now, uint32_t sampled_at, float temperature, bo
    if (++recovery_count_ < 2) { error_ = Error::Recovering; return; }
    state_ = temperature >= threshold_ ? State::Alert : State::Normal;
  } else if (state_ == State::Alert) {
-   if (temperature <= threshold_ - kHysteresis) state_ = State::Normal;
+   if (temperature < threshold_ - kHysteresis) state_ = State::Normal;
  } else if (temperature >= threshold_) {
    state_ = State::Alert;
  }
}

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\revert> .\scripts\test.cmd; $LASTEXITCODE
FAIL line 80: p.snapshot(4000).state == State::Normal
FAIL line 83: p.snapshot(i * 1000).state == (i == 5 ? State::Normal : State::Alert)
FAIL 114 assertions; 2 failures; synthetic host tests
1
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\revert> git revert --no-edit HEAD; $LASTEXITCODE; git --no
-pager log --oneline -3; git status --short
[recover-hysteresis 92e71dc] Revert "FAULT-hysteresis: seed one deliberate teaching defect"
Date: Sun Sep 13 12:17:13 2026 -0400
1 file changed, 1 insertion(+), 1 deletion(-)
0
92e71dc (HEAD -> recover-hysteresis) Revert "FAULT-hysteresis: seed one deliberate teaching defect"
299f10f (tag: FAULT-hysteresis, origin/fault-hysteresis) FAULT-hysteresis: seed one deliberate teaching defect
1fe0a4f (tag: CP09, origin/main, origin/HEAD, main) CP09: record the chapter starting state
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\revert> Start-Transcript -Path 'C:\Users\user\Documents\AI
-Embedded-Book\M304recovered01.txt'; .\scripts\test.cmd; $rc=$LASTEXITCODE; $rc; if($rc -eq 0){$env:ARDUINO_CLI='C:\Users\user\
Tools\aiem\arduino-cli.exe'; .\scripts\build-firmware.cmd; $LASTEXITCODE}; Stop-Transcript
Transcript started, output file is C:\Users\user\Documents\AI-Embedded-Book\M304recovered01.txt
PASS 114 assertions; 0 failures; synthetic host tests
0
Sketch uses 362508 bytes (12%) of program storage space. Maximum is 2883584 bytes.
Global variables use 54952 bytes (16%) of dynamic memory, leaving 272728 bytes for local variables. Maximum is 327680 bytes.
0
Transcript stopped, output file is C:\Users\user\Documents\AI-Embedded-Book\M304recovered01.txt
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m03-v01\revert>

```

SS07-06. Revert the seeded bad commit and verify recovered behavior.

8. Lab LAB07: review and recover

Submit a review note rejecting the unrelated candidate edit, a resolved conflict README, and before/after evidence for the seeded fault. Keep each exercise in its designated disposable repository so restoring an uncommitted sentence cannot destroy work you meant to revert as history.

For each recovery, write four short fields: starting state; intended data to preserve; chosen command or edit; observed result. For restore, the intended loss is the explicitly unwanted uncommitted edit. For revert, preserve existing history plus a corrective commit. For conflict resolution, preserve both useful meanings and remove the markers.

You pass when final content meets the requirement, relevant tests show the intended recovery, and useful work remains accounted for. A blank diff alone is insufficient: you might have discarded the requested feature along with the fault. Compare final behavior with the original brief.

For a final self-check, explain the recovery before touching the keyboard. If your explanation is “get back to where it worked,” ask which file version, which commit, and which useful work must remain. That small pause turns a vague rescue attempt into a reviewable operation. You should be able to predict whether the command changes working content, staging, or history.

Completion checklist

- ☐ I began each exercise from a known state in the correct practice repository.
- ☐ I reviewed changes by purpose and rejected the unrelated edit.
- ☐ I resolved the README conflict for meaning and removed all markers.
- ☐ I distinguished discarding an uncommitted edit from reverting a commit.
- ☐ I reproduced the seeded failure and saved actual recovery evidence.

Review questions with answers

1. Does a branch protect every uncommitted edit? No. Branches name history; working changes may remain present while you switch when Git permits it. Start clean or preserve and account for edits explicitly.

2. Why not accept both conflict sides automatically? Both fragments may duplicate or contradict each other. Resolve the intended meaning, inspect the finished document or program, and run relevant checks.

3. Why use revert for a shared committed error? It records a new corrective event while retaining existing history. Restore primarily addresses file contents and staging; it is not a substitute for an explicit historical reversal.

Transfer exercise

Choose a small change you expect to make soon. Write its branch purpose, review criteria, and recovery plan before editing. Name the state you would preserve if the experiment failed. This is the foundation for using an agent confidently on a project whose history matters.

8. Markdown as an Engineering Notebook

What you will be able to do

You will turn working notes into a README that another person can follow, preview the result, repair a broken link, and distinguish a command from its expected output. CP01 and the Git habits from Chapters 5-7 are prerequisites. Use CP02 for this chapter's linked README exercise because it supplies the hardware reference. The deliverable is a readable README plus a short record of the setup and link checks you actually performed. No board is required.

CP01-CP04 are deliberately small baselines: the host command prints one fixed JSON reading with `synthetic: true` and `temperature 25.00`. It does not parse fixtures or implement the later policy. The full project architecture is the destination; this chapter documents the current baseline honestly.

Plain text with a second view

A useful notebook lets you return to a project without reconstructing its history from memory. Markdown is a practical choice because its source is ordinary text: an editor can open it, Git can show its changes, and a person can read it before any publishing system is available. A renderer interprets some punctuation as structure. You therefore work with two views of one document: the source you edit and the formatted page your reader sees.

That distinction matters in an engineering project. A shell command may look correct in the editor but become an awkward paragraph in the preview. A heading may look decorative while failing to create a useful document outline. A link may have an informative label while pointing to a file that moved last week. Reviewing only the source misses these faults. Reviewing only the preview can hide a command that copied incorrectly or a path that depends on your computer.

Keep the notebook close to the code. In the learner project, `README.md` lives at the Sensor Monitor root, alongside `firmware`, `host`, `scripts`, and `docs`. It answers the first questions: what the project does, what is needed, how to run a check, and where to read further. It does not need every design decision or every log. Those records can have their own files once they contain useful information.

Use the editor's Markdown preview to compare source and rendering. In VS Code, open the Command Palette and search for **Markdown: Open Preview to the Side**. The exact placement of controls can change between versions; the command name and document result are the useful landmarks. The official [VS Code Markdown guide](#) describes the editor's Markdown support. Figure SS08-01 identifies the source structures; SS08-02 pairs them with the rendered view.

The screenshot shows the Visual Studio Code interface with a workspace named 'm04-101 (Workspace)'. The Explorer sidebar on the left shows the file structure, including a 'sensor-monitor' folder with subfolders 'docs', 'firmware', 'fixtures', 'host', and 'scripts', along with files like '.gitignore', 'CHECKPOINT.md', 'dependencies.json', and 'README.md'. The main editor window displays the 'README.md' file with the following content:

```

1  # Sensor Monitor
2
3  This baseline prints a fixed synthetic report on the computer. Shared policy and fixture parsing arrive at CP05;
   hardware is optional.
4
5  ## Run synthetic replay
6
7  From the Sensor Monitor root, the folder containing scripts, docs, host and firmware:
8
9  1. Read [setup](docs/setup.md) and [dependencies](dependencies.json).
10 2. Copy only the command below, without a terminal prompt.
11 3. Record its actual exit and output separately.
12
13 ```powershell
14 .\scripts\replay.cmd
15 ```
16
17 Expected: exit 0 and a fixed synthetic JSON sample with temperature_c 25.00.
18
19 ## Understand the evidence
20
21 Read the [hardware reference](docs/hardware.md) before changing target settings.
22
23 | Action | Establishes | Limitation |
24 | --- | --- | --- |
25 | Actual CP02 host replay | Builds and prints the fixed synthetic report | No fixture parsing, assertions or electrical
   observation |
26 | Actual firmware cross-build | Compiles/links the selected FQBN | No upload, connected board or physical measurement |
27

```

The status bar at the bottom indicates 'Screen Reader Optimized', 'Ln 1, Col 1', 'Spaces: 4', 'UTF-8', 'LF', and 'Markdown'.

SS08-01. Record the evidence table for the lesson workflow.

Explorer

m04-101 (Workspace)

sensor-monitor

docs

firmware

fixtures

host

scripts

.gitignore

CHECKPOINT.md

dependencies.json

README.md

README.md > # Sensor Monitor

1

Sensor Monitor

2

3

This baseline prints a fixed synthetic report on the computer. Shared policy and fixture parsing arrive at CP05; hardware is optional.

4

5

Run synthetic replay

6

7

From the Sensor Monitor root, the folder containing scripts, docs, host and firmware:

8

9

1. Read [setup](docs/setup.md) and [dependencies](dependencies.json).

10

2. Copy only the command below, without a terminal prompt.

11

3. Record its actual exit and output separately.

12

13

```powershell

14

.\scripts\replay.cmd

15

```

16

17

Expected: exit 0 and a fixed synthetic JSON sample with temperature_c 25.00.

18

19

Understand the evidence

20

21

Read the [hardware reference](docs/hardware.md) before changing target settings.

22

23

| Action | Establishes | Limitation |

24

| --- | --- | --- |

25

| Actual CP02 host replay | Builds and prints the fixed synthetic report | No fixture parsing, assertions or electrical observation

This baseline prints a fixed synthetic report on the computer. Shared policy and fixture parsing arrive at CP05; hardware is optional.

Run synthetic replay

From the Sensor Monitor root, the folder containing scripts, docs, host and firmware:

1. Read [setup](#) and [dependencies](#).

2. Copy only the command below, without a terminal prompt.

3. Record its actual exit and output separately.

.\scripts\replay.cmd

Expected: exit 0 and a fixed synthetic JSON sample with temperature_c 25.00.

Understand the evidence

Read the [hardware reference](#) before changing target settings.

Action	Establishes	Limitation
Actual CP02 host replay	Builds and prints the fixed synthetic report	No fixture parsing, assertions or electrical observation
Actual firmware cross-build	Compiles/links the selected FQBN	No upload, connected board or physical measurement

SS08-02. Preview the rendered command, hardware reference, and evidence table.

Give the page a navigable structure

Use one top-level heading for the document title, then second-level headings for its major sections. Add third-level headings only when a section contains real subsections. A heading names the material that follows it. “Run synthetic replay” helps a returning reader more than “More information.” Consistent levels also help navigation and accessibility in the exported book and course.

```
# Sensor Monitor

A small firmware and host-test project for learning a verified workflow.

## Requirements

## Run synthetic host replay

## Build firmware

## Troubleshooting
```

This is a structure to populate, not a finished README. Delete unused headings rather than collecting empty categories. A short complete document is easier to trust than a long list of promises. As the project grows, add a section when a reader has a question that cannot be answered in the existing structure.

Use bullets for parallel facts and numbered steps when order matters. “Install the tools, open the project root, run the replay command” is a sequence. “Host tests, synthetic replay, optional board deployment” names alternatives with different evidence. Do not hide ordered work inside a long paragraph, and do not turn every sentence into a bullet. The format should expose the relationships between the ideas.

Leave a blank line around headings, lists, tables, and code blocks. This makes the source easier to inspect and reduces surprises across renderers. Use ordinary paragraphs for explanation. Bold text can draw attention to a required starting directory or an evidence limitation, but a page where everything is emphasized has no hierarchy.

Show commands as commands

Inline backticks identify a short filename, state name, or code expression: `README.md`, `FAULT`, or `temperature >= threshold`. For a command that a learner should copy, use a fenced block. The opening fence can include a language label such as `powershell`, `cpp`, or `text`. A language label helps rendering; it does not execute or validate the content.

From the Sensor Monitor project root, the supplied Windows replay entry point is:

```
.\scripts\replay.cmd
```

Explain the environment before the block. This command runs the supplied, reviewable Windows wrapper from PowerShell. Read the wrapper and follow the setup route documented

for your machine. The wrapper configures its process environment; the lesson does not require changing PowerShell execution policy. If the command fails, record the message and inspect the documented dependency and working-directory checks.

The command's working directory is part of the instruction. A relative path beginning `.\scripts` is meaningful only when the current directory contains that folder. Do not include a prompt such as `PS C:\...>` in a copyable block. It is useful visual context in a screenshot, but it is not part of the command. Keep output in a separate text block or describe the acceptance condition in prose.

For this chapter, the acceptance condition is that the replay command completes successfully and its output matches the documented fixed synthetic report. Your exact output belongs in your own evidence record. Do not paste a fabricated “all tests passed” line into the README as though it came from your run. A clearly labeled expected result is helpful; a pretend observation is harmful. This difference becomes central in Chapter 9.

A firmware cross-build has a different purpose:

```
.\scripts\build-firmware.cmd
```

The wrapper preserves the selected target configuration. A successful build demonstrates that the source compiles and links with the recorded toolchain and options. It does not show that a board was connected or that a physical sensor produced readings. Put that limitation beside the command, where the reader is about to interpret its result.

Use links as part of the procedure

A Markdown link has descriptive text and a destination. At the repository root, this example points into the project documentation:

```
Read the [hardware reference](docs/hardware.md) before changing board settings.
```

The words explain why a reader would follow the link. “Click here” does not. Relative links keep the project portable: the same relationship can work in another checkout without the author's absolute computer path. The destination is interpreted relative to the file containing the link, so a link in `docs/handoff.md` has a different starting point from one in `README.md`.

Suppose the `handoff` needs to link back to the root README. Its relative destination is `../README.md`: go up one directory, then select the file. Suppose two documents both live under `docs`. Their link can use the other document's filename directly. Before creating a path with several `..` segments, inspect the tree and make sure the document organization itself is still understandable.

Test links by following them in the rendered view, as illustrated by SS08-03. Confirm that the opened document is the intended one, not merely a file with a similar name. Check capitalization consistently even when the current operating system tolerates a mismatch. A

project that works on one case-insensitive filesystem can reveal errors on another system or in a publishing build.

The screenshot shows the Visual Studio Code interface. On the left, the Explorer pane shows a workspace named 'm04-102 (Workspace)' with a folder 'sensor-monitor' containing subfolders 'docs', 'firmware', 'fixtures', 'host', and 'scripts', along with files '.gitignore', 'CHECKPOINT.md', 'dependencies.json', and 'README.md'. The main editor shows the 'README.md' file with the following content:

```

1 # Sensor Monitor – CP02
2
3 The current baseline prints one fixed synthetic report.
  Hardware is optional.
4
5 ## Prerequisites and starting directory
6
7 Read [setup](docs/setup.md) and the exact [dependency
  record](dependencies.json).
8 Open the project root containing scripts, docs, host and
  firmware.
9
10 ## First check
11
12 ```powershell
13 .\scripts\replay.cmd
14 ```
15
16 Expected criterion: exit 0 and one JSON sample with type
  sample, synthetic true and temperature_c 25.00. Save
  actual output separately. This stage has no fixture
  parsing or native assertion suite.
17
18 ## Firmware compilation
19
20 ```powershell
21 .\scripts\build-firmware.cmd
22 ```
23
24 The maintained wrapper preserves the dependency FQBN.
  Compilation does not prove upload or physical operation.
25

```

On the right, a preview of the 'Toolchain setup' document is shown. It contains the following text:

The recorded example toolchain is Arduino CLI 1.5.1, ESP32 core 3.3.11, and native ARM64 LLVM-MinGW 20260908 / Clang 23.1.1 on Windows 11 ARM. Native compiler and Arduino CLI archive URLs and SHA-256 values are in [dependencies.json](#). Check each download before extraction:

```

Get-FileHash -Algorithm SHA256 .\downloaded-tool.zip
Expand-Archive -LiteralPath .\downloaded-tool.zip -
DestinationPath "$env:USERPROFILE\Tools\aiem"

```

Use the actual downloaded archive filename. The LLVM archive has its own top-level directory; keep that directory. The project scripts detect `$env:USERPROFILE\Tools\aiem\llvm-mingw-20260908-ucrt-aarch64\bin\clang++.exe`, or use `$env:CXX` explicitly. Add the folder containing `arduino-cli.exe` to the current PowerShell PATH or set `$env:ARDUINO_CLI`. Record `clang++ --version`, `arduino-cli version`, and `arduino-cli core list` in your own evidence.

Install the core through the official package index (network access required):

```

arduino-cli core update-index --additional-urls
https://espressif.github.io/arduino-
esp32/package_esp32_index.json
arduino-cli core install esp32:esp32@3.3.11 --
additional-urls https://espressif.github.io/arduino-
esp32/package_esp32_index.json

```

SS08-03. Follow the README link to the toolchain setup document.

Links into headings can be convenient, but heading identifiers depend on the renderer. If the target heading changes, revisit incoming links. For a beginner-facing setup path, linking to a small document is often more durable than relying on a deeply nested heading anchor. External links should name the source and land on the relevant page rather than on a generic search result.

Images use related syntax with an exclamation mark. Supply alternative text that conveys the relevant information, and keep a caption when provenance or interpretation needs explanation. A screenshot of passing host tests needs a caption naming the test and checkpoint. It should not be labeled “working hardware.” The image helps a reader recognize a state; the selectable command and explanation must still teach the step without requiring them to read small pixels.

Tables and checklists earn their space

A table is useful when each row can be compared using the same columns. For example:

Action	Establishes	Does not establish
Host test	Behavior for supplied cases	Electrical behavior
Synthetic replay	Shared logic on controlled input	Sensor measurement accuracy
Firmware build	Compatibility with selected build inputs	Successful upload or wiring

The vertical bars separate cells and the hyphen row separates the header. GitHub's table syntax is documented in its [table-formatting guide](#). Keep rows short enough to read on a narrow screen. A complex troubleshooting explanation usually belongs in prose with a descriptive heading, not in a table containing paragraphs inside every cell.

A checklist can record the progress of a procedure. It does not validate its own claims. Checking “build passed” because the command was attempted confuses an action with an outcome. Use separate items when the distinction matters: command completed, exit status inspected, output reviewed, and evidence saved. Avoid a checklist of vague qualities such as “good documentation.” Replace it with something observable, such as “every setup link opens the intended file.”

Do not rely on a checkbox as the sole explanation. A checked item should point to the evidence or be accompanied by a concise note. If the check is a plan for later work, leave it unchecked and label the document as a plan. This is a simple way to avoid converting hope into history during a hurried update.

Worked example: turn terse notes into a README

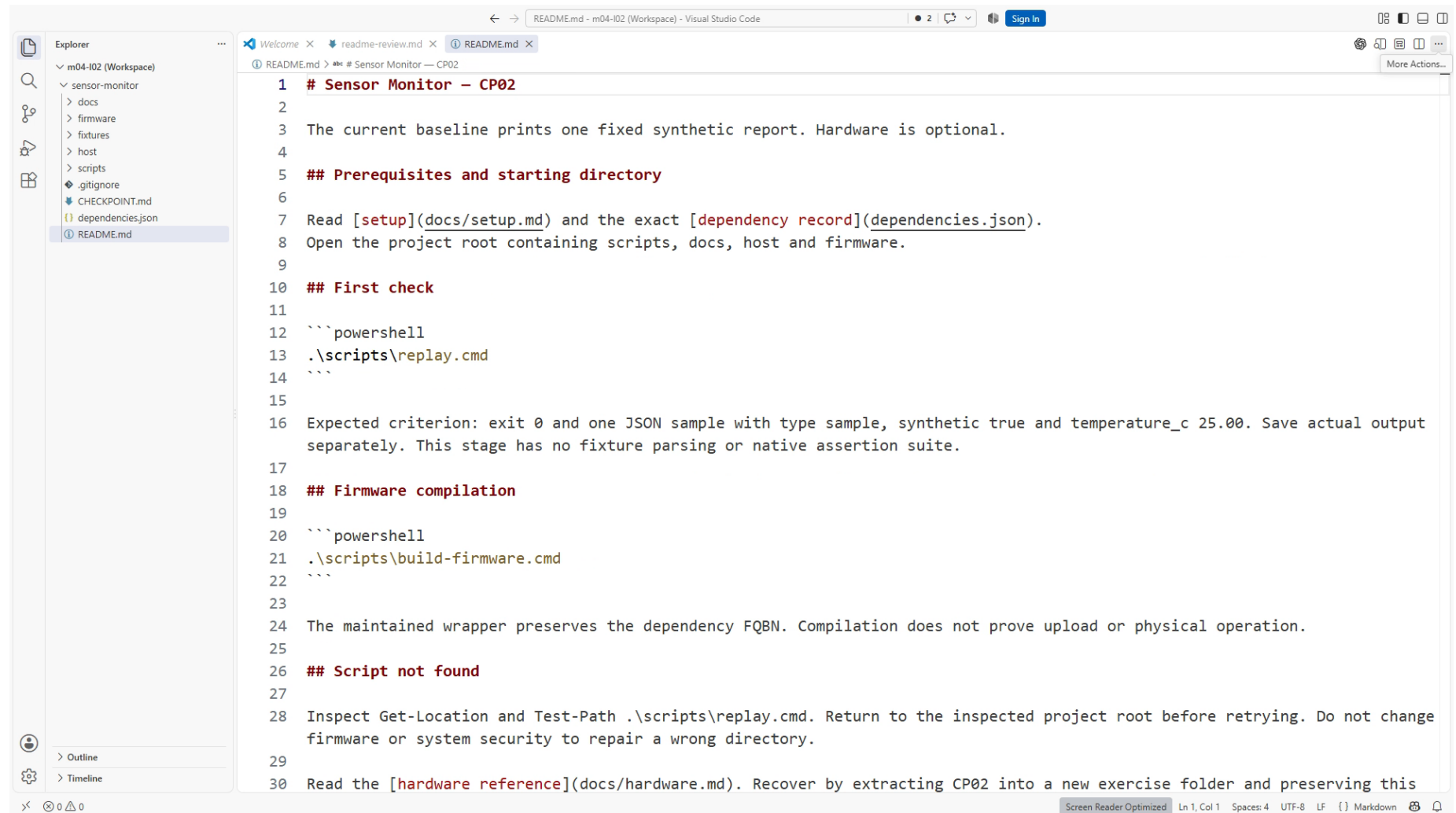
Consider these deliberately incomplete notes:

```
sensor demo. clang installed. run tests. board build later.
```


They may remind the author what happened yesterday, but they do not tell another person what to do. Begin by identifying missing meaning. “Sensor demo” lacks an outcome. “Clang installed” lacks a version reference and installation route. “Run tests” lacks a working directory and command. “Later” names neither a requirement nor a current result.

Rewrite the current purpose first: “This checkpoint prints a fixed synthetic report on the host and provides a minimal sketch for cross-compilation.” State that no physical board is needed. Shared policy and fixture-driven replay are later milestones, beginning at CP05. This gives readers an accurate model before asking them to install anything.

Next, refer to the release's tool setup and dependency record, rather than copying an incidental machine path. Add the replay command in a labeled block and describe success by the process result and documented fixed synthetic output. Host assertion tests are introduced at CP07; CP01-CP04 use replay and build checks. Link to the hardware reference for the exact cross-build target. Add a short troubleshooting entry for running from the wrong folder. SS08-04 shows the sections that make this path complete.



The screenshot shows the Visual Studio Code interface with a workspace named 'm04-102 (Workspace)'. The Explorer sidebar on the left shows the file structure: 'sensor-monitor' containing 'docs', 'firmware', 'fixtures', 'host', 'scripts', '.gitignore', 'CHECKPOINT.md', 'dependencies.json', and 'README.md'. The main editor window displays the 'README.md' file with the following content:

```
1 # Sensor Monitor - CP02
2
3 The current baseline prints one fixed synthetic report. Hardware is optional.
4
5 ## Prerequisites and starting directory
6
7 Read [setup](docs/setup.md) and the exact [dependency record](dependencies.json).
8 Open the project root containing scripts, docs, host and firmware.
9
10 ## First check
11
12 ```powershell
13 .\scripts\replay.cmd
14 ```
15
16 Expected criterion: exit 0 and one JSON sample with type sample, synthetic true and temperature_c 25.00. Save actual output
17 separately. This stage has no fixture parsing or native assertion suite.
18
19 ## Firmware compilation
20
21 ```powershell
22 .\scripts\build-firmware.cmd
23 ```
24
25 The maintained wrapper preserves the dependency FQBN. Compilation does not prove upload or physical operation.
26
27 ## Script not found
28
29 Inspect Get-Location and Test-Path .\scripts\replay.cmd. Return to the inspected project root before retrying. Do not change
30 firmware or system security to repair a wrong directory.
31
32 Read the [hardware reference](docs/hardware.md). Recover by extracting CP02 into a new exercise folder and preserving this
```

The status bar at the bottom indicates 'Screen Reader Optimized', 'Ln 1, Col 1', 'Spaces: 4', 'UTF-8', 'LF', and 'Markdown'.

SS08-04. Inspect the finished README for setup, build, expected output, and recovery guidance.

Finally, follow your instructions from top to bottom. Do not fill gaps silently from memory. If you have to locate an unmentioned compiler, change directory, or open a different reference, the README needs that information. A second learner can provide valuable feedback, but you can already find many problems by treating yourself as a reader with only the written steps available.

Ask an agent to review the document with a bounded request: “Identify missing prerequisites, ambiguous working directories, and links that do not resolve in this checkout. Propose edits only to the README.” Its review is a useful second perspective. You still inspect the rendered result and the actual commands. Fluent documentation can be wrong just as fluent code can be wrong.

LAB08: add a recoverable troubleshooting path

Allow about 25-40 minutes. Extract CP02 into a new `lab08` exercise folder or use your existing preserved copy. Do not overwrite earlier work. Open the Sensor Monitor root and note the starting Git status if it is a repository.

1. Read the README from the perspective of a person returning after a month. Mark one unclear prerequisite or working-directory assumption.
2. Add a **Troubleshooting** section with a “Script not found” entry. Explain how to confirm the project root and locate `scripts/replay.cmd` before retrying.
3. Include a relative link to `docs/hardware.md`, plus a sentence explaining when it is needed. Keep host replay and firmware compilation as separate actions.
4. Open the Markdown preview. Follow the relative link, return to the README, and copy the replay command from the preview into the intended terminal.
5. Run the documented check when the required tools are available. Save the actual outcome, checkpoint, command, and any remaining setup problem in your exercise notes.
6. Review the diff. Confirm that your change improves navigation or reproduction and does not introduce machine-specific paths or unrelated source edits.

Submit the edited README and the short check record. If tool setup is incomplete, your formatting and link checks can still be complete, but the build-path check remains open. Record the exact missing dependency; do not award yourself a passing execution result merely because the command is present.

Failure and recovery: the plausible broken link

Seed the fault only in your exercise copy by changing `docs/hardware.md` to `doc/hardware.md`. The label still looks sensible. The rendered page may still look polished. Following it reveals the defect. Compare the destination with the actual directory name, restore the correct path, and follow it again. Preserve both the cause and the correction in your notes.

A second fault is a command written as an ordinary sentence. A reader may copy extra punctuation or miss a line break. Move the command into a fenced powershell block, keep the

explanation outside it, and test the copy operation. Formatting is functional here: it helps preserve the exact input needed for a reproducible procedure.

Completion check and review answers

You are finished when the README explains purpose, prerequisites, working directory, commands, expected outcomes, evidence limits, and a useful recovery path; its preview is readable; its links resolve; and the recorded execution outcome is honest.

Why inspect both source and preview? The source exposes exact syntax and paths; the preview exposes what the reader will see and copy. Neither view replaces the other.

Does a language label run a code block? No. It helps rendering and tells the reader what kind of text it contains. Execution is a separate action with a separate result.

Why prefer relative project links? They express relationships within a checkout instead of depending on an author's machine location. They still need testing from the document that contains them.

Can a screenshot replace a command? No. It can illustrate the result, but the instruction should remain searchable, selectable, and understandable without small image text.

What does a successful firmware build prove? It supports a software build claim for the recorded inputs. Physical operation, wiring, and sensor accuracy require different evidence.

Transfer to your own work

Choose one small project you already understand. Write a six-section README using purpose, prerequisites, first check, expected result, references, and troubleshooting. Ask someone else to follow it if available, or repeat it from a fresh exercise folder. Record every assumption you had to supply from memory. Those assumptions are the most useful material for the next revision.

Optional deeper practice: review the README's Git diff after a week of changes. Can you distinguish changed behavior from improved wording? If an important command changed, connect that edit to the release or dependency change that justified it. Your notebook is now part of the engineering workflow, not an accessory added at the end.

Sources and figures

Primary references: [VS Code Markdown support](#) and [GitHub tables](#), accessed September 12, 2026. Commands and file names refer to the accompanying Sensor Monitor release. Figure IDs: SS08-01 source, SS08-02 preview, SS08-03 link navigation, SS08-04 finished README.

Screenshots document actual demonstrations only; the written examples above are instructional text, not transcripts of an agent run.

9. Build a Useful Project Knowledge Base

What you will be able to do

You will separate intended behavior, sourced facts, design choices, and actual observations; connect them with a few useful links; and repair a contradiction without erasing its history. Start with CP02 and the README skills from Chapter 8. Your deliverable is a coherent set of project documents that a returning engineer or a new agent session can inspect. The exercise uses manufacturer documentation and synthetic software evidence, so physical hardware is optional.

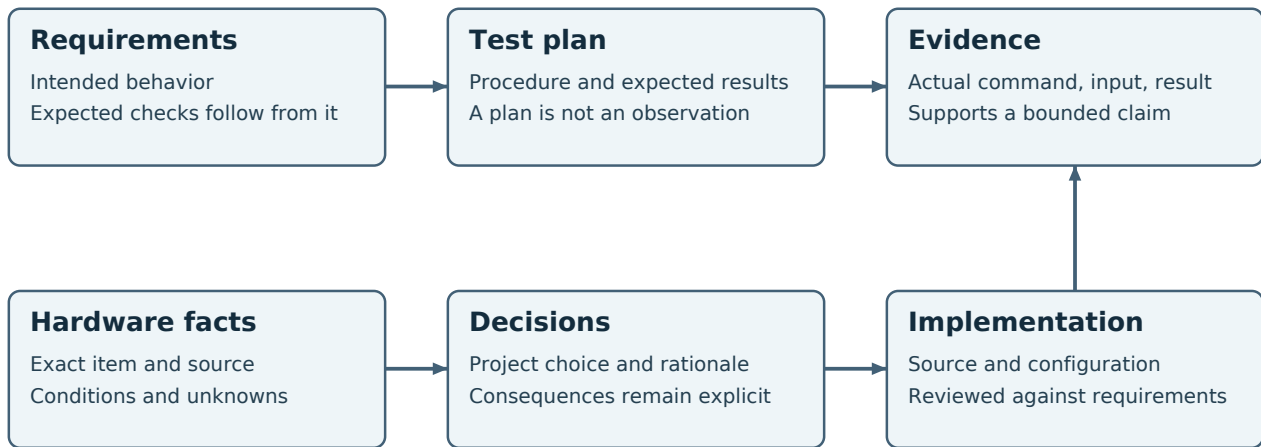
Four kinds of information

A note saying “the monitor clears at 27 degrees and works” compresses several questions into one vague assertion. Is 27 degrees a requirement? Is it the observed behavior of the current code? Which starting state was used? Was the input synthetic? Did anyone run the program? Good project context separates these questions before asking an AI system to reason from them.

A **requirement** describes intended behavior that can be checked. A **hardware fact** describes the selected device according to a named source. A **decision** records a choice made by the project and its rationale. A **result** records an observation from a particular procedure. These categories can support one another, but they are not interchangeable. A requirement does not establish that the implementation meets it, and a product page does not establish that your wiring is correct.

For Sensor Monitor's planned capstone, “enter ALERT at or above 28.0 °C” is a software requirement. The selected example board's identity belongs in the hardware reference. Keeping future threshold changes in RAM is a project decision with consequences for reset behavior. Output produced by running a specified input is a result. Figure D08 maps these relationships: requirements guide tests, hardware facts constrain implementation, decisions explain choices, and results support bounded claims.

D08 · Requirements, facts, decisions, and evidence



Simplified conceptual relationships · original course artwork · no runtime or physical result implied

D08. Requirements, sourced facts, decisions and actual evidence support different parts of a reviewable engineering result.

The useful knowledge base is small enough to inspect. CP02 supplies `README.md`, `docs/project-brief.md`, and `docs/hardware.md`. Read those first. As you write meaningful content in this chapter, add the other records; do not expect them to exist in the untouched checkpoint. The resulting set uses five documents under `docs`: `requirements.md`, `hardware.md`, `decisions.md`, `evidence.md`, and `handoff.md`. Create a separate test-plan document only if the plan becomes difficult to navigate inside the existing structure. The filenames are less important than giving each important statement an identifiable home.

Start with a project brief

The README or an opening section of `docs/requirements.md` can hold the brief. Explain purpose, scope, and acceptance. CP02 prints a fixed synthetic host report and provides a minimal sketch for the documented target. Shared policy and fixture-driven replay begin at CP05. Name that later architecture as a planned milestone, preserving the distinction between current behavior and the destination.

Describe exclusions only when they protect understanding of the project. A reader might reasonably expect a sensor monitor to include a wireless dashboard or physical measurements. State that the core exercise uses terminal output and does not require board ownership. Avoid a long list of every feature imaginable. Scope is useful when it prevents a plausible misunderstanding, not when it becomes a catalogue of things the project has never considered.

Give requirements stable identifiers when later records need to refer to them. For example, R04 identifies alert entry at the threshold, while R05 identifies hysteresis. A stable identifier

survives improved wording and helps a reviewer follow the relationship between a test and its purpose. It is not a substitute for clear language: “meets R04” means little unless R04 itself states an observable condition.

Write boundary behavior explicitly. “Alert when hot” leaves equality undefined. “While in NORMAL, enter ALERT when a valid input temperature is at least 28.0 °C” tells a tester what to expect for 27.9, 28.0, and 28.1. Hysteresis also needs a starting state: while already in ALERT, clearing at or below 27.0 is different from deciding the state of a first sample. Record the condition that makes the requirement meaningful.

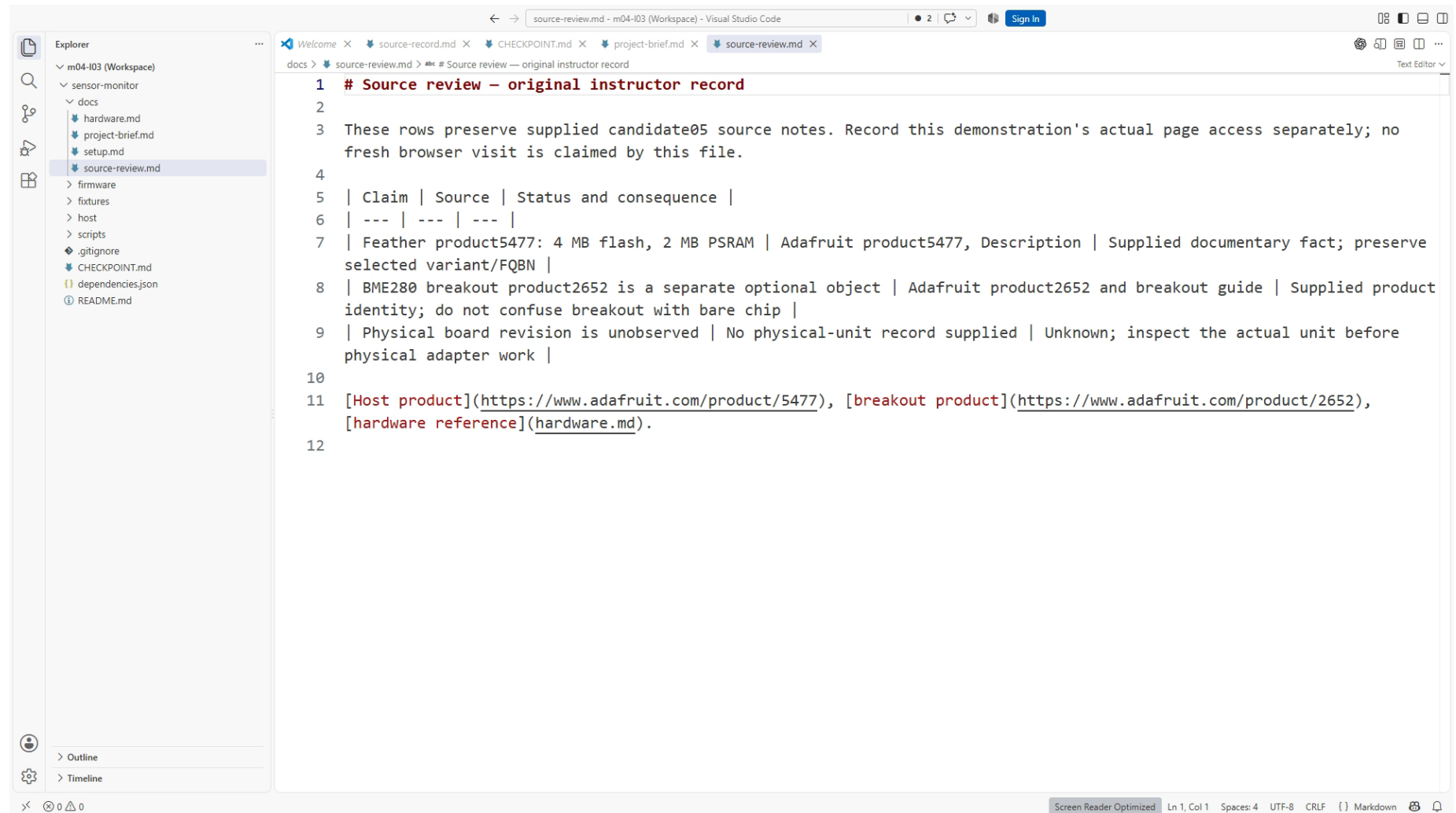
Identify the exact hardware reference

Family names are useful for searching, but they are weak evidence for a board-specific setting. “ESP32-S3 Feather” can be insufficient if products differ in memory configuration or circuitry. The documented example target is the Adafruit ESP32-S3 Feather product 5477, described as the 4 MB flash / 2 MB PSRAM variant. The [manufacturer product page](#) and the [board guide](#) are starting references for checking identity and board-specific details.

The example sensor is the Adafruit BME280 breakout, product 2652. Its [product page](#) identifies the breakout; its [guide](#) supplies board-level usage context. A chip datasheet answers different questions, such as register definitions and measurement specifications. Keep breakout, chip, and host-board sources distinct so a claim about one is not silently transferred to another.

Use a compact table containing the item, exact identity, source, source section or revision, and open questions. “Revision not established from supplied material” is a valid entry. An empty cell invites a later reader or agent to assume that the detail was forgotten rather than intentionally unresolved. A confidence label should explain its basis: “documented in product identification” is more useful than a naked percentage.

Do not fill pin mappings from a familiar board in memory. A source may establish the existence of a connector without establishing every initialization step needed by its switched power circuit. Record what the source actually supports and the consequence for the project. If exact revision evidence is missing, say what must be checked before the optional physical build. SS09-02 focuses on this connection between identity, source, and uncertainty.



The screenshot shows the Visual Studio Code interface with a workspace named 'm04-103 (Workspace)'. The Explorer sidebar on the left shows a file tree with folders 'sensor-monitor' and 'docs'. The 'docs' folder contains 'hardware.md', 'project-brief.md', 'setup.md', and 'source-review.md'. The 'source-review.md' file is selected and open in the editor. The editor shows the following content:

```
1 # Source review – original instructor record
2
3 These rows preserve supplied candidate05 source notes. Record this demonstration's actual page access separately; no
  fresh browser visit is claimed by this file.
4
5 | Claim | Source | Status and consequence |
6 | --- | --- | --- |
7 | Feather product5477: 4 MB flash, 2 MB PSRAM | Adafruit product5477, Description | Supplied documentary fact; preserve
  selected variant/FQBN |
8 | BME280 breakout product2652 is a separate optional object | Adafruit product2652 and breakout guide | Supplied product
  identity; do not confuse breakout with bare chip |
9 | Physical board revision is unobserved | No physical-unit record supplied | Unknown; inspect the actual unit before
  physical adapter work |
10
11 [Host product](https://www.adafruit.com/product/5477), [breakout product](https://www.adafruit.com/product/2652),
  [hardware reference](hardware.md).
12
```

The status bar at the bottom indicates 'Screen Reader Optimized', 'Ln 1, Col 1', 'Spaces: 4', 'UTF-8', 'CRLF', and 'Markdown'.

SS09-02. Record sourced board and memory facts while leaving physical revision unobserved.

Cite enough to find the claim again

A useful source entry lets someone relocate the evidence after the conversation has disappeared. Record the publisher, title, direct URL, relevant section or page, access date, and revision if the source exposes one. For a local supplied excerpt, identify its origin and preserve the nearby conditions that affect interpretation. A number without its unit or operating conditions is often worse than no number because it looks precise while being incomplete.

Distinguish a quotation from your interpretation. You can paraphrase a source briefly and then write a separate consequence, such as “therefore keep this target option in the build wrapper.” The consequence is your engineering reasoning. Readers should be able to disagree with that reasoning while still agreeing on what the source says. Do not copy whole manuals into a repository merely to make the context larger; link to the source and preserve only material you have permission to redistribute.

Dates matter because a live guide may change. An access date tells a future reviewer when you checked it; it does not prove that the page was revised that day. Record “revision not stated” when appropriate. The project's tested tool versions belong in its release or dependency record, not as a guess inferred from the latest download page.

When sources conflict, first check whether they describe the same item and condition. A chip limit and a breakout-board input recommendation may differ without either being wrong. A different product variant may explain an apparent memory mismatch. If the discrepancy remains unresolved, preserve both references and keep the affected claim open. Averaging conflicting values or choosing the more convenient one is not a resolution.

Decisions explain why the project looks this way

A decision record should be short enough to read during an edit. Give it a title, status, date, context, choice, alternatives considered, and consequence. For example, “Use host replay for the required core demonstration” explains that the course needs reproducible software checks without physical equipment. The consequence is an explicit limitation: replay does not establish electrical behavior, actual sensor readings, or device timing.

A good decision can later be replaced. Mark the old record superseded and link to the new one if the rationale changes. Do not quietly rewrite history so that every past choice appears inevitable. The purpose is to help the next engineer understand which assumptions remain active and which no longer apply.

Separate project choices from manufacturer facts. “Use a 28.0 °C teaching threshold” is not a BME280 calibration specification. “Keep configuration in RAM” is not a limitation imposed by the processor. Mixing these categories can cause an agent to defend a convenient default as if it were an immutable hardware constraint.

The decision record is also a place to capture a tradeoff without overbuilding. A single maintained build wrapper may be easier for learners than several platform-specific commands copied into every document. That choice creates a responsibility to keep the wrapper readable

and record its dependencies. It does not require a new framework or a complex document approval system.

A test plan is not a test result

The test plan describes what you will do and what would count as success. Write expected outcomes before looking at the current implementation. For threshold entry, an expected state at exactly 28.0 follows from the requirement. If you derive the expected value by copying the code's comparison, the test may repeat the same mistake as the implementation.

A result adds the actual command, checkpoint or source identity, input source, environment, exit status, relevant output, date, and interpretation. Name CP02's fixed sample; identify a fixture when the later adapter actually reads one. Record limitations. "Host test with synthetic input" is clear; "sensor verified" is too broad. A cross-build and host test can both pass while a physical connection is absent.

Use evidence pointers rather than burying every line of output in a central document. `docs/evidence.md` can summarize a run and link to a saved log in the appropriate evidence location. Preserve the actual log, including a failure that led to a correction, when it matters to the explanation. A summary is for navigation; the underlying output is what lets another reviewer check the claim.

Treat an AI statement such as "tests should pass" as a prediction. Even "tests passed" needs a visible command/result or an accessible saved record. Ask which command ran, against which source, and where the output is. If the record is missing, change the statement to "not established by this record" and rerun the relevant check when possible. Do not manufacture a log to match a confident answer.

Worked example: connect behavior to evidence

As a look ahead to the later capstone, suppose a proposed change adjusts threshold entry. This planning example does not require CP02 to contain the later assertion suite. Begin with R04 in `requirements.md`: while in NORMAL, valid temperatures at or above 28.0 °C enter ALERT. Startup and FAULT recovery have their own requirements; this example does not bypass them. In `decisions.md`, note that the threshold is a teaching default and is not a claim about sensor calibration. In `hardware.md`, retain the exact example target without inventing a causal link between that target and the chosen threshold.

The planned host cases use inputs immediately below, exactly at, and immediately above the boundary. Each expected outcome is derived from R04. A result record then reports what actually happened when the supplied test wrapper ran. If the equality case fails, the record should say so. That failure can be valuable evidence that the test detects a meaningful defect.

Now consider a proposed summary: "The board alerts correctly at 28.0." This statement exceeds the evidence if only host tests ran. Replace it with "The shared policy's host test accepted the specified equality behavior for the recorded source and synthetic cases," if that is what the run actually establishes. The revised statement is more informative because it names both the success and its boundary.

SS09-03 illustrates the distinction between a decision and a result. The text of an actual run may differ from this example. A teaching caption should describe the visible record, not rewrite it into the result an author hoped to obtain.

SS09-03. Record the decision, plan, actual replay result, and handoff together.

LAB09: repair conflicting context

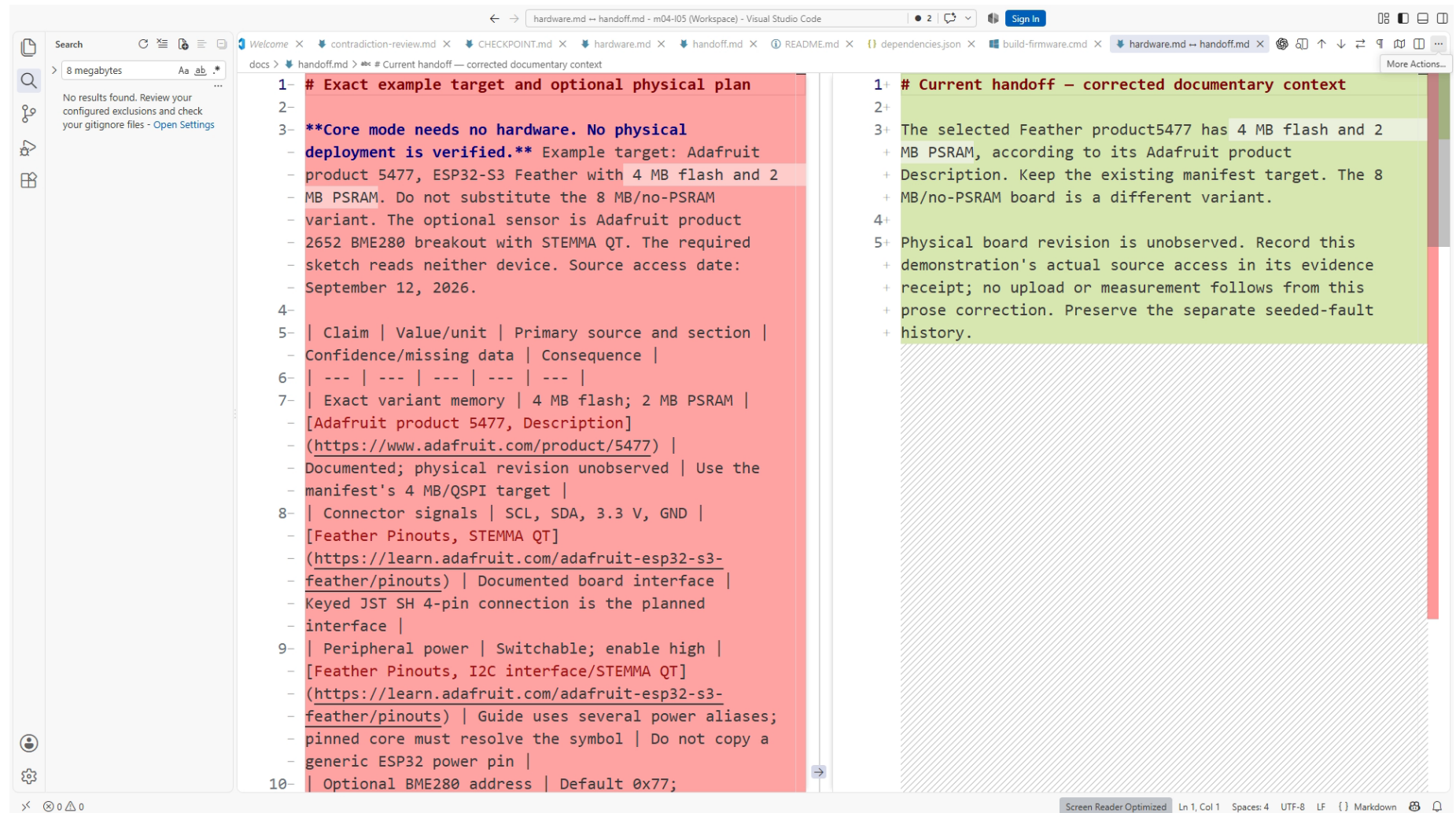
Allow 35-50 minutes. Use a new CP02 exercise folder. Read README.md, docs/project-brief.md, and docs/hardware.md. Create or update docs/requirements.md, docs/decisions.md, docs/evidence.md, and docs/handoff.md only with the populated content taught in this chapter. Label future threshold requirements planned; CP02 does not implement them. Preserve the supplied version before editing. Create an exercise note that records the starting checkpoint and the documents you inspect.

1. In a disposable copy of docs/hardware.md, seed the claim “The selected product 5477 has 8 MB flash.” Label the exercise note as an intentionally seeded fault so it cannot be mistaken for reference truth.
2. Open the manufacturer's product page for product 5477. Confirm that the page identifies the intended product, then locate its memory description.
3. Compare the seeded sentence with the selected target's source. Correct the active hardware reference to the documented 4 MB flash / 2 MB PSRAM identity, adding the source and relevant location.
4. Search the exercise documents for the stale 8 MB claim and the product identifier. Update references that actually describe this selected target. Do not globally replace unrelated numbers.
5. Add a brief decision or correction note explaining that a variant mismatch caused the contradiction. Keep the exercise's fault record separate from current guidance.
6. Read the resulting linked documents in order and review the diff. Confirm that the change corrected the fact without inventing a board-observed result.

Submit the corrected context and an evidence note with source, claim, correction, and remaining unknowns. The exercise tests source resolution, not possession of the physical board. A source that cannot be accessed is an open dependency; use the release's permitted source notes and record that limitation rather than fabricating a browsing result.

Failure and recovery: stale context survives the correction

A common repair changes the main hardware table but leaves a contrary instruction in a handoff. The next agent reads the handoff first and repeats the old assumption. Recovery therefore includes searching for the identifier and stale claim, deciding which records are current, and linking historical records to the corrected source. SS09-04 focuses on that reviewable correction.



SS09-04. Compare the current hardware and handoff documents as separate context sources.

Do not solve this by deleting all old records. An old result can remain accurate for the old configuration. Mark its scope and prevent it from presenting itself as current guidance. A present-tense hardware reference and a dated historical test record serve different purposes and can coexist without conflict when their labels are clear.

Handoff without the old conversation

End a work session with a short handoff: current checkpoint, active task, reviewed changes, last relevant checks with evidence pointers, unresolved questions, and the next bounded action. “Continue improving everything” is not a useful next task. “Resolve the missing document revision before extracting pin constraints” gives the next session a concrete uncertainty to address.

Arbitrary Markdown files are not automatically loaded by every agent. A well-written `handoff.md` still needs to be named or linked from guidance the tool discovers. In the next chapter, `AGENTS.md` directs Codex to the right project references. Context organization and context discovery are related but separate jobs.

Completion check, questions, and transfer

Complete the chapter when each important statement has a clear category, source-dependent claims have traceable references, the seeded contradiction is resolved across current documents, and observed results remain separate from plans. Explain D08 aloud using one requirement, one hardware fact, one decision, and one result.

Is “use 28.0 °C” a hardware fact? No. Here it is a project requirement or decision. It does not describe sensor accuracy or calibration.

Can a test plan say that a feature works? It can state the intended acceptance condition. An actual result is needed to support an implementation claim.

What if a revision is missing? Record that it is not established by the available material and identify what source would resolve it. Do not invent a revision.

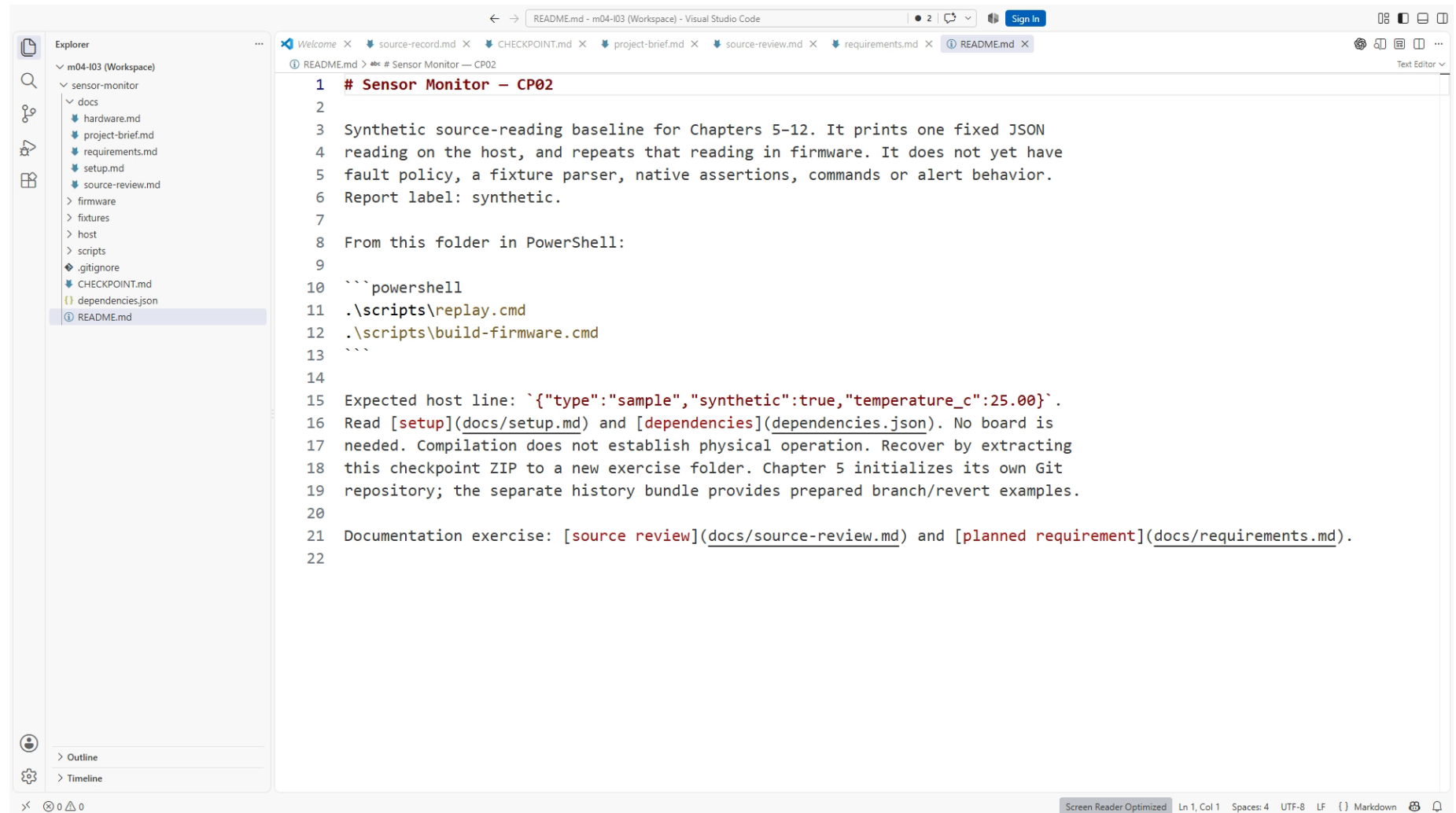
Why keep historical failed results? They can show the original defect and the effect of the correction. Their date, source identity, and status prevent confusion with current guidance.

Does saving a Markdown file make every agent read it? No. Discovery is host-specific. Name or link the relevant file and verify that the intended workflow uses it.

For transfer, choose one disputed fact in your own project. Trace it to the exact source, record the consequence, and find every current document that repeats it. Then write a handoff that another session could use without your conversation. This is a small knowledge base doing useful work: reducing the number of assumptions that must be rediscovered.

Sources and figures

Primary hardware references: Adafruit [product 5477](#), [ESP32-S3 Feather guide](#), [product 2652](#), and [BME280 guide](#). Source checks are dated in the accompanying release evidence. Project behavior comes from the Sensor Monitor requirements and matching test artifacts. Figures: SS09-01 document tree, SS09-02 sourced facts, SS09-03 decision/result, SS09-04 contradiction correction, and D08 context relationships.



SS09-01. Inspect the populated document tree and its linked project context.

10. Write AGENTS.md for an Embedded Project

What you will be able to do

You will write concise project guidance, check its discovery scope, and test whether a fresh Codex session follows it during a bounded task. Start with CP02, the context documents from Chapter 9, and working project replay/build commands. Your deliverable is CP03-style guidance plus a behavior record. A successful result requires observable actions; the agent's summary of its instructions is only a diagnostic clue.

A fresh CP03 download contains the project brief and hardware reference, with replay/build commands but no assertion suite. Your own continued Chapter 9 copy may also contain the additional notebook documents you wrote. Link only files present in the working copy: use `docs/project-brief.md` for the CP03 behavior task, and add requirements/decision/evidence links when those populated files exist.

Guidance that survives a conversation

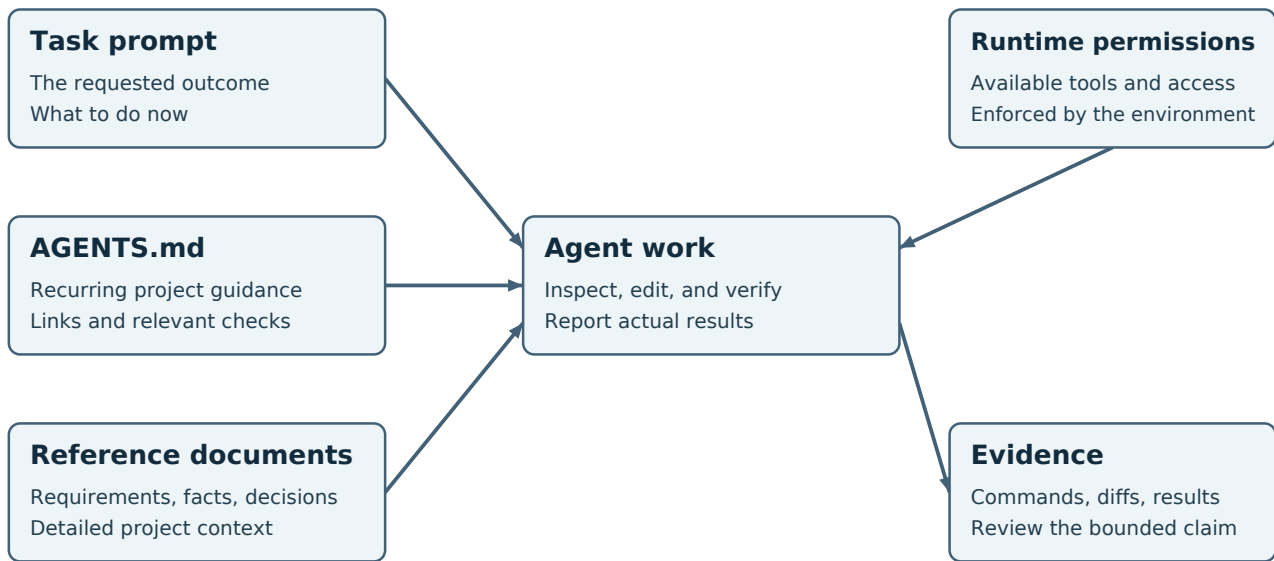
A good engineering brief answers what to do now. Project instructions answer what usually matters here. Without that distinction, every request grows into a repeated introduction: use the shared policy, consult the exact board reference, keep synthetic output labeled, run the maintained tests, and report the evidence. Repetition is tiresome and makes it easy to omit a constraint when the task becomes interesting.

AGENTS.md gives those recurring expectations a maintained home in the repository. For Sensor Monitor, it can explain the project purpose, direct the agent to current requirements and hardware facts, identify the supported validation commands, and define what a completion report must contain. The current task still supplies the specific requested change. The reference documents still hold detailed engineering information.

Think of the file as a signpost with working agreements. It should help the agent choose the right files and checks without copying the whole repository into persistent instructions. A ten-page hardware manual inside AGENTS.md becomes another manual to keep synchronized. A short instruction to read `docs/hardware.md` before changing a board-specific setting preserves one source of truth.

Guidance does not grant capability. A sentence saying “run the compiler” cannot install a missing compiler, authorize a blocked action, or make a tool available. A sentence saying “do not access other folders” is useful intent but is not a filesystem security boundary. Actual runtime permissions, approval policy, and available tools govern what operations the environment can perform. Figure D09 separates task prompt, project guidance, reference files, and runtime controls.

D09 · Project guidance and runtime permissions



Simplified conceptual relationships · original course artwork · no runtime or physical result implied

D09. Task instructions and project references guide work; runtime permissions independently constrain available tools and access.

Filename, location, and discovery

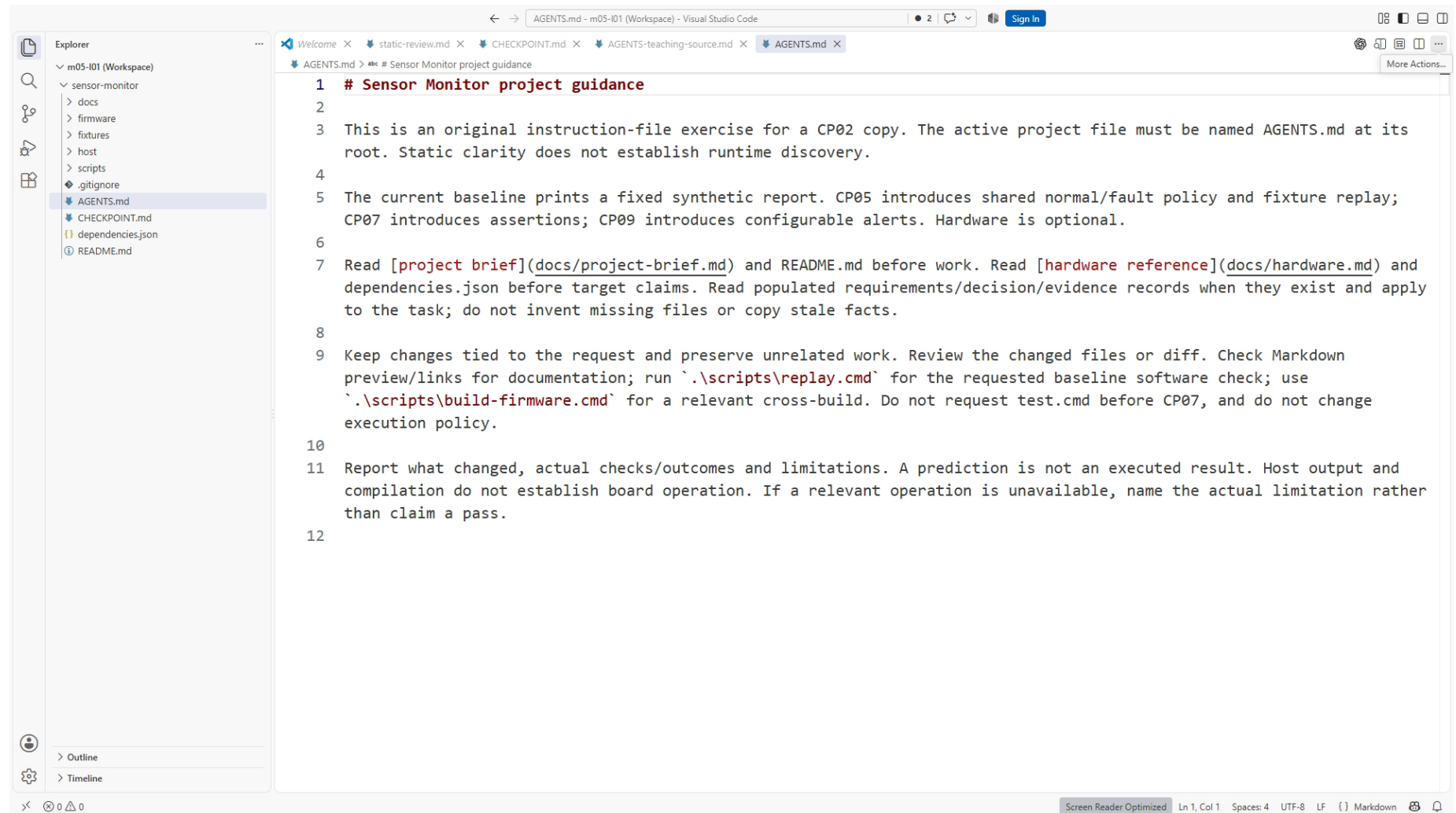
Use the exact filename `AGENTS.md` at the Sensor Monitor project root. Confirm the editor has not created `AGENTS.md.txt`, and do not confuse it with `.agents/skills`, which has a different purpose in Chapter 11. Inspect the folder that the agent actually opened. A correct file in a neighboring checkout is irrelevant to the active task.

The official [OpenAI AGENTS.md guide](#), checked September 12, 2026, describes Codex's startup instruction chain: global guidance is considered first; project discovery walks from the project root toward the current working directory. At each directory it selects at most one applicable instruction file, preferring `AGENTS.override.md` over `AGENTS.md`, followed by configured fallback names. More local guidance appears later. Empty files are skipped, and the combined project-document budget is 32 KiB by default. Restart or begin a new run after an instruction change when verifying the chain.

These are documented Codex rules, not a universal promise about every AI editor. The course uses a root file to keep the initial lesson simple. If the installed interface reports a different active workspace or does not show the intended file, diagnose that condition before assessing whether the wording is effective. A discovery failure and an instruction-quality failure need different repairs.

Avoid changing global settings merely to complete this exercise. Global preferences can affect unrelated projects, and an existing override may explain an unexpected result. Inspect the relevant scope with the available interface, preserve existing configuration, and keep the

exercise local. SS10-02 focuses on the exact filename and location. The laboratory records the installed interface and version so future readers know which behavior was actually tested.



The screenshot shows the Visual Studio Code interface with the Explorer view on the left and the AGENTS.md file open in the editor. The Explorer view shows the workspace structure for 'm05-101 (Workspace)', including a 'sensor-monitor' folder with subfolders like 'docs', 'firmware', 'fixtures', 'host', and 'scripts'. The 'AGENTS.md' file is selected in the Explorer. The editor view shows the content of 'AGENTS.md', which is a project guidance document. The document is titled '# Sensor Monitor project guidance' and contains several paragraphs of text. The text is numbered 1 through 12. The status bar at the bottom indicates 'Screen Reader Optimized', 'Ln 1, Col 1', 'Spaces: 4', 'UTF-8', 'LF', and 'Markdown'.

```
1 # Sensor Monitor project guidance
2
3 This is an original instruction-file exercise for a CP02 copy. The active project file must be named AGENTS.md at its
  root. Static clarity does not establish runtime discovery.
4
5 The current baseline prints a fixed synthetic report. CP05 introduces shared normal/fault policy and fixture replay;
  CP07 introduces assertions; CP09 introduces configurable alerts. Hardware is optional.
6
7 Read [project brief](docs/project-brief.md) and README.md before work. Read [hardware reference](docs/hardware.md) and
  dependencies.json before target claims. Read populated requirements/decision/evidence records when they exist and apply
  to the task; do not invent missing files or copy stale facts.
8
9 Keep changes tied to the request and preserve unrelated work. Review the changed files or diff. Check Markdown
  preview/links for documentation; run `.\scripts\replay.cmd` for the requested baseline software check; use
  `.\scripts\build-firmware.cmd` for a relevant cross-build. Do not request test.cmd before CP07, and do not change
  execution policy.
10
11 Report what changed, actual checks/outcomes and limitations. A prediction is not an executed result. Host output and
  compilation do not establish board operation. If a relevant operation is unavailable, name the actual limitation rather
  than claim a pass.
12
```

SS10-02. Inspect the saved guidance and exact AGENTS.md at the Explorer root.

Write rules that tell an engineer what to do

“Write excellent embedded code” expresses a preference without defining a reviewable action. “After changing shared policy or parser code, run the host-test wrapper and report its exit result” is operational. It names the trigger, the action, and the evidence. Instructions work best when a reviewer could inspect a completed task and decide whether each expectation was satisfied.

Similarly, replace “be careful with hardware” with a boundary tied to this project: the required path uses synthetic host input and firmware compilation; physical upload and wiring belong to a separately chosen optional lab. The purpose is to prevent a software result from being described as a board measurement and to avoid accidental expansion of the requested task.

Specify a starting directory for commands. Use the maintained wrappers rather than embedding compiler flags in several documents. Explain which checks apply to which changes. A wording-only edit to the README may require link and preview checks; changing C++ policy requires stage-appropriate checks, including host tests once available; changes affecting firmware integration need the cross-build. Requiring every expensive check after every punctuation fix creates noise and makes meaningful omissions harder to notice.

Use strong requirements sparingly. If every sentence begins “always” or “never,” ordinary exceptions become contradictions. “Run relevant checks before claiming completion; report any check you could not run” is more useful than “always run all tests” when a task cannot access the required dependency. Reporting a limitation is part of a truthful completion report, not permission to silently omit required work.

Worked example: a concise root file

The following is an instructional pattern using the reference project's paths. Compare it with the maintained AGENTS.md shipped in your checkpoint before making an edit. The released file is the source of truth for the exact command set.

```
# Sensor Monitor project guidance

Read the stage README: CP01-04 print a fixed synthetic report;
CP05 adds shared policy and fixture replay. Firmware is cross-compiled
for a documented example target; physical operation is unverified.

## Start here

Read README.md and docs/project-brief.md for the current stage.
Read docs/requirements.md when that populated document exists.
Read docs/hardware.md before changing board-specific settings.
Read docs/decisions.md and docs/evidence.md when those populated files exist.

## Work and validation

Keep changes bounded to the requested outcome and inspect the diff.
Run commands from the project root using the maintained scripts.
At CP07 or later, after shared C++ changes run scripts/test.cmd.
At earlier checkpoints, use the available replay/build checks.
```

```
After firmware integration changes, also run scripts/build-firmware.cmd.
Use scripts/replay.cmd; name the synthetic fixture at CP05 or later.
```

```
## Finish
```

```
Report changed behavior, relevant checks, actual results, and limitations.
Keep expected outcomes separate from observed output.
Do not describe host replay or a build as physical board verification.
Physical wiring and upload are outside the required core exercise.
```

This pattern is intentionally small. It does not define a new agent, launch a subagent, or introduce a new access-control system. It also does not attempt to anticipate every future task. Read each line and ask what recurring error it prevents. If you cannot identify one, the line may belong in an explanation rather than in persistent guidance.

At CP03, the available Windows replay invocation is:

```
.\scripts\replay.cmd
```

The firmware wrapper is separate:

```
.\scripts\build-firmware.cmd
```

Use these with the supplied reviewable wrappers and documented setup. The current Windows wrappers configure their process environment without changing PowerShell execution policy. At CP07 and later, `.\scripts\test.cmd` supplies host assertion tests as well. If execution fails, preserve the actual message and resolve the demonstrated environment issue. Adding “ignore restrictions” to Markdown does not fix a missing dependency or grant authorization.

Keep detailed facts in their own documents

Suppose the hardware reference changes because a source check resolves a board variant. If `AGENTS.md`, `README`, handoff, and three prompts each contain a copied target string, all five copies can become stale. Prefer a single hardware reference and a maintained build wrapper, with instructions directing the agent to both when relevant.

The same principle applies to requirements. Persistent guidance should say where behavior is specified, while the requirement states the exact boundary. Otherwise, a task that legitimately changes a threshold can conflict with an old number embedded in the guidance. You want a reviewer to see that the requirement changed deliberately, not to spend time discovering an invisible duplicate rule.

This separation also makes a fresh-session test more informative. If the prompt repeats the entire requirement and every command, success may show only that the agent followed the prompt. A test that asks for a bounded change while relying on repository guidance can reveal whether the signposts actually work. You can then inspect the file reads, chosen commands, and final evidence report.

Nested rules: use only when a real scope differs

A larger repository may contain firmware, a web interface, and an unrelated tool with different validation commands. Local guidance can be useful there. Sensor Monitor is small, so begin with one root file. Adding nested instruction files merely to demonstrate sophistication makes it harder to know which rule applies.

If you later need a nested rule, document its reason and test from the intended working directory. Do not assume that opening a deep file means the current directory has changed. Identify the actual task scope before reasoning about discovery. A rule in an unused sibling directory does not automatically belong to the active chain.

When two instructions conflict, remove the accidental disagreement at its source if possible. For example, root guidance that says “use the maintained test wrapper” and a stale local note that says “skip tests to save time” undermine the workflow. Diagnose whether the local file is a deliberate specialization or obsolete guidance. Preserve a useful distinction; delete or revise redundant contradictory wording in the exercise copy.

Verify behavior, not reassurance

Begin with a diagnostic request such as: “Identify the project instruction sources and the documents relevant to a shared policy change. Do not edit files.” Compare its response with the actual folder and files. This can reveal a missing instruction file or a surprising override. It is not yet a successful behavior test.

For the real test, use a new session and a small recoverable task. At CP03, ask it to clarify the README's synthetic-input explanation using the project brief and complete the documented replay check. Do not paste the command into the task. The intended sequence is context inspection, a bounded documentation edit, diff review, actual replay, and an accurate result report. At CP07 or later, adding a meaningful test for an implemented behavior is an alternative; keep that later task separate from the CP03 case.

Do not score a particular sentence style. The agent may explain the work differently and still satisfy the engineering outcome. Score observable decisions: correct requirement, appropriate scope, actual command, valid result interpretation, and preserved unrelated files. An announced plan to run tests is not execution. A final claim needs a corresponding command/result or saved evidence record.

Save the prompt, checkpoint, interface version, instruction-file identity, actual observations, changed-file list, commands/results, and verdict. SS10-03 shows the fresh task; SS10-04 shows command use. If the run fails, keep the record and state the cause that the evidence supports. This is an evaluation of a workflow, so an honest failed test is useful material for improving it.


```

1  # Actual fresh-session task excerpts - M05-L02
2
3  Coordinator-selected excerpts from preserved actual-events.jsonl; not a simulated agent transcript.
4
5  Exact prompt: Clarify the README's synthetic-input explanation using the project brief. Keep the edit bounded and
   complete the documented replay check.
6
7  Session: 01a09c42-3af7-7821-a57f-97c0791eabae
8
9  ## Actual commands and exits
10
11  Command: "C:\\WINDOWS\\System32\\WindowsPowerShell\\v1.0\\powershell.exe" -Command 'Get-Content README.md; Get-Content
   docs/project-brief.md; Get-Content docs/hardware.md; Get-Content dependencies.json; git status --short'
12  Native exit: 1
13
14  Command: "C:\\WINDOWS\\System32\\WindowsPowerShell\\v1.0\\powershell.exe" -Command ".\\scripts\\replay.cmd"
15  Native exit: 0
16  Output:
17  {"type":"sample","synthetic":true,"temperature_c":25.00}
18
19
20  Command: "C:\\WINDOWS\\System32\\WindowsPowerShell\\v1.0\\powershell.exe" -Command 'Get-Content README.md -TotalCount
   10; git diff --check'
21  Native exit: 1
22
23  Command: "C:\\WINDOWS\\System32\\WindowsPowerShell\\v1.0\\powershell.exe" -Command ".\\scripts\\replay.cmd;
   '$replayExit = $LASTEXITCODE; Get-Content README.md -TotalCount 8; exit $replayExit'
24  Native exit: 0
25  Output:
26  {"type":"sample","synthetic":true,"temperature_c":25.00}
27  # Sensor Monitor â€” CP03
28

```

SS10-03. Begin a fresh task using the repository instructions and exact source context.

```

41 # Sensor Monitor "CP03
51 ## Actual final answer
57 {"type": "sample", "synthetic": true, "temperature_c": 25.00}
58 ...
59
60 The folder is not a Git repository, so a Git diff/check was unavailable.
61
62 ## Independent coordinator verification
63 {
64     "exit": 0,
65     "scope": "actual host replay; no physical observation",
66     "stdout_file": "C:\\Users\\user\\Documents\\AI-Embedded-Book\\capture-evidence-m04-m05-v02\\m05-l02-
baseline\\coordinator-replay.txt",
67     "root": "C:\\Users\\user\\Documents\\AI-Embedded-Book\\capture-practice-m04-m05-v02\\m05-l02-baseline\\sensor-
monitor",
68     "command": ".\\scripts\\replay.cmd",
69     "reviewer": "coordinator",
70     "stdout_sha256": "06FAA66CC0CEE41E42420E376CAE091745F52B7015A52DE576C1EBD9F434F25C",
71     "checkpoint": "CP03",
72     "utc": "2026-09-13T19:35:35.8384268Z"
73 }
74
75 ## Source inventory comparison
76 Changed source: README.md SHA256 BB299C81DDDB438C22AAAACC82CB7146CF5EF3F3BA2970AD378B2405758425D7
77
78 Generated replay build artifacts are separate from source edits. This ZIP copy has no Git metadata; the observed Git
status/diff failures are repository-state limitations, not a replay failure.
79 No physical result or human learner test is claimed. A fresh task does not prove absence of all global/application
context.

```

SS10-04. Inspect the recorded command result and changed README inventory after execution.

LAB10: test a fresh session

Allow 40-60 minutes, excluding account or dependency waiting. Extract CP03 into a new exercise folder. Preserve your normal project and record the starting README and instruction file before the agent begins. Use a bounded reporting-documentation task; the CP03 exercise does not require later capstone features or assertion tests.

1. Read the root `AGENTS.md`, `README`, project brief, and validation wrappers. Identify the instructions relevant to source-grounded documentation and replay.
2. Start a genuinely new task or session in that exercise root. A continued conversation carrying your previous explanations is not the intended test.
3. Submit: “Clarify the README's synthetic-input explanation using the project brief. Keep the edit bounded and complete the documented replay check.”
4. Observe the actual work. Record whether it consults the relevant references, confines the documentation edit, uses the documented check, and interprets the result accurately.
5. Independently inspect the diff and the replay output. Preserve your evidence outside the source files under review so a log update does not obscure the code change.
6. If guidance was not followed, diagnose discovery, wording, conflict, or missing capability. Make the smallest justified repair and repeat in another fresh session.

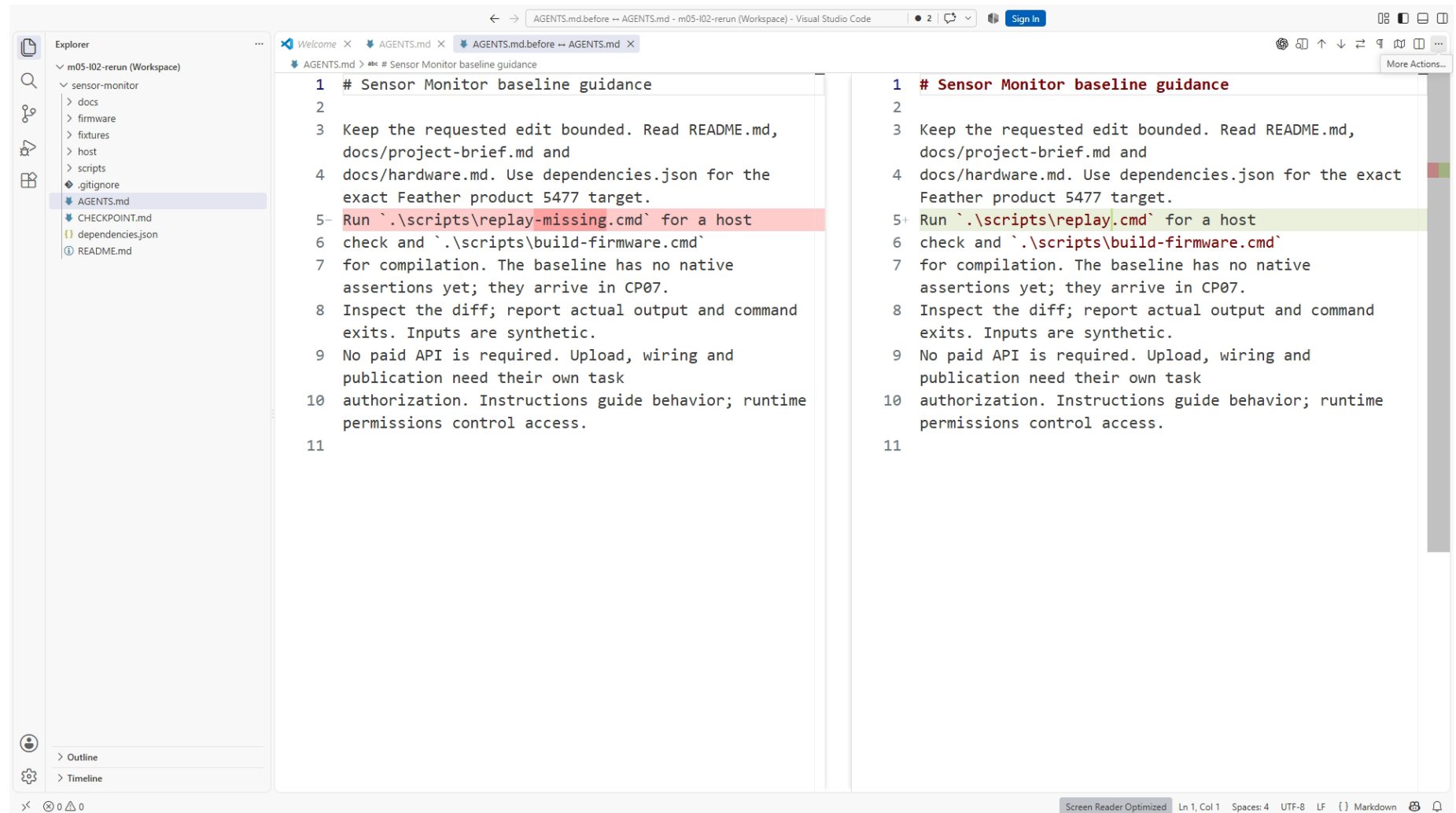
Submit the final guidance, the task record, and the independent review. A passing verdict requires behavior evidence. If your account reaches a limit or the tool cannot execute, label that run blocked or incomplete and use the supplied reference materials for continued reading; do not convert a static inspection into an observed pass.

Failure and recovery: broad guidance produces a broad edit

Imagine that the guidance says “Improve the project whenever you see an opportunity.” During the README clarification, an agent reorganizes unrelated files. The result may look polished but becomes harder to review. In the exercise copy, inspect the changes and preserve any work you need before returning to the documented start. Replace the broad sentence with a bounded expectation: keep edits tied to the requested outcome and report worthwhile unrelated suggestions separately.

Repeat the same task in a fresh session and inspect whether the scope improves. Do not infer causation from one attractive response alone. The repair is supported when the changed wording addresses the observed ambiguity and the repeated task meets the acceptance criteria. If the broad edit continues, investigate other active instructions and whether the task itself invites expansion.

Another failure is an instruction to run a command that no longer exists. This is a maintenance defect, not evidence that the agent is disobedient. Open the referenced path, compare it with the current scripts, correct the guidance, and repeat the relevant test. SS10-05 records the repaired guidance and rerun behavior without pretending the initial attempt succeeded.



SS10-05. Compare the preserved seeded guidance with the corrected replay command.

Completion check and review answers

Complete the chapter when the file is concise, correctly located, internally consistent, linked to current references, and supported by a fresh-session behavior record. If execution remains unavailable, the document can be ready while the behavior gate remains open. Keep those statuses separate.

Is AGENTS.md access control? No. It is guidance. Runtime permissions and approvals control actual operations; the file cannot grant capabilities or securely fence off data.

Does it create a subagent? No. It describes project expectations for the agent reading it. Agent definitions and delegation are separate capabilities.

Why start a fresh session? A continued conversation can carry instructions that hide discovery defects. A fresh task better tests whether repository guidance supplies what is needed.

Is an instruction summary sufficient evidence? No. It can diagnose discovery. A behavior test must show the relevant actions and results during a real task.

What should happen when a required check cannot run? The task should report the exact limitation, retain the evidence, and avoid a passing claim. Resolve the environment or repeat the check later.

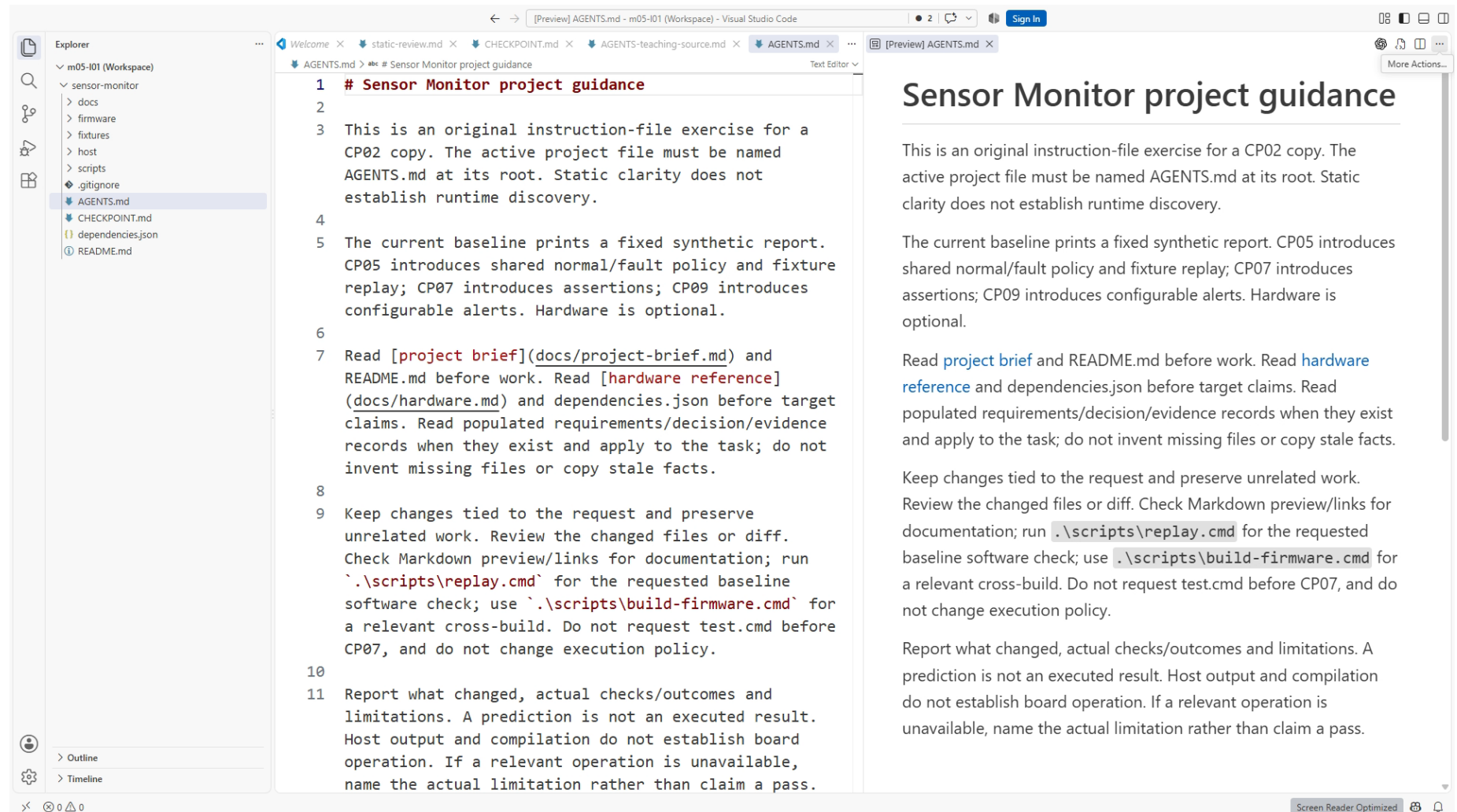
Transfer to another project

Select three expectations you repeat in your own coding requests. Convert each into a trigger, action, and evidence requirement. Put detailed facts in referenced documents. Test one small task in a fresh session without repeating the commands in its prompt. The result gives you a practical measure of whether the guidance reduces repeated explanation while preserving review quality.

Optional deeper note: maintain a tiny regression set for project guidance after substantial tool or instruction changes. Include one routine code task and one documentation-only task. The second case checks that the guidance does not turn a harmless wording correction into unnecessary broad work. Evaluate the finished artifacts and actual actions rather than rewarding particular phrases.

Sources and figures

Host-specific discovery guidance: [OpenAI AGENTS.md documentation](#), accessed September 12, 2026. The project's actual instruction and command files govern its released examples. Figures: SS10-01 guidance content, SS10-02 location, SS10-03 fresh task, SS10-04 observed command, SS10-05 repair/rerun, and D09 guidance versus reference context and permissions. This chapter defines the test; only the accompanying actual run record can establish its result.



The screenshot displays the Visual Studio Code interface. On the left, the Explorer sidebar shows a workspace named 'm05-101 (Workspace)' containing a folder 'sensor-monitor' and files 'AGENTS.md', 'CHECKPOINT.md', 'dependencies.json', and 'README.md'. The main editor area shows the 'AGENTS.md' file with the following content:

```
1 # Sensor Monitor project guidance
2
3 This is an original instruction-file exercise for a
4 CP02 copy. The active project file must be named
5 AGENTS.md at its root. Static clarity does not
6 establish runtime discovery.
7
8 The current baseline prints a fixed synthetic report.
9 CP05 introduces shared normal/fault policy and fixture
10 replay; CP07 introduces assertions; CP09 introduces
11 configurable alerts. Hardware is optional.
12
13 Read [project brief](docs/project-brief.md) and
14 README.md before work. Read [hardware reference]
15 (docs/hardware.md) and dependencies.json before target
16 claims. Read populated requirements/decision/evidence
17 records when they exist and apply to the task; do not
18 invent missing files or copy stale facts.
19
20 Keep changes tied to the request and preserve unrelated
21 work. Review the changed files or diff. Check Markdown
22 preview/links for documentation; run
23 `.\scripts\replay.cmd` for the requested baseline
24 software check; use `.\scripts\build-firmware.cmd` for
25 a relevant cross-build. Do not request test.cmd before
26 CP07, and do not change execution policy.
27
28 Report what changed, actual checks/outcomes and
29 limitations. A prediction is not an executed result.
30 Host output and compilation do not establish board
31 operation. If a relevant operation is unavailable,
32 name the actual limitation rather than claim a pass.
```

The right sidebar shows a preview of the document content, titled 'Sensor Monitor project guidance'. The preview text is as follows:

This is an original instruction-file exercise for a CP02 copy. The active project file must be named AGENTS.md at its root. Static clarity does not establish runtime discovery.

The current baseline prints a fixed synthetic report. CP05 introduces shared normal/fault policy and fixture replay; CP07 introduces assertions; CP09 introduces configurable alerts. Hardware is optional.

Read [project brief](#) and README.md before work. Read [hardware reference](#) and dependencies.json before target claims. Read populated requirements/decision/evidence records when they exist and apply to the task; do not invent missing files or copy stale facts.

Keep changes tied to the request and preserve unrelated work. Review the changed files or diff. Check Markdown preview/links for documentation; run `.\scripts\replay.cmd` for the requested baseline software check; use `.\scripts\build-firmware.cmd` for a relevant cross-build. Do not request test.cmd before CP07, and do not change execution policy.

Report what changed, actual checks/outcomes and limitations. A prediction is not an executed result. Host output and compilation do not establish board operation. If a relevant operation is unavailable, name the actual limitation rather than claim a pass.

SS10-01. Preview the bounded guidance, purpose, links, and commands.

11. Turn a Repeated Procedure into a Skill

What you will be able to do

You will package a useful source-checking procedure, write its trigger description, and evaluate relevant, unrelated, and incomplete-input requests. Start with CP03 and the source discipline from Chapters 9-10. Your deliverable is the datasheet-to-constraints skill and a test record. The skill produces a traceable constraints table; it does not invent missing hardware information or claim physical verification.

Choose the right kind of reuse

You have now written task prompts, project documents, and recurring instructions. A skill serves another purpose: it packages a procedure that should be applied to a recognizable kind of task. The procedure can include instructions, focused references, and optional deterministic helpers. It is worth packaging when you have already performed the work and can describe both good output and common failure cases.

Use a prompt for a one-off request. Use `AGENTS.md` for recurring project expectations. Use a script when an operation should behave deterministically, such as checking whether required fields are present in a CSV. Use a skill when a repeated task needs judgment, source selection, a sequence of steps, and an output contract. These forms can cooperate without becoming copies of one another.

For Sensor Monitor, extracting constraints from a primary hardware document involves judgment: identify the exact part, preserve operating conditions, distinguish a chip from a breakout, and explain what is still unknown. A script can check that every row has a source field, but it cannot by itself establish that a cited page supports a pin mapping. The skill guides that reasoning and exposes its result for human review.

Do not begin by making a universal “embedded engineering” skill. That description spans firmware design, test writing, wiring, purchasing, debugging, and documentation. It is difficult to trigger correctly and nearly impossible to evaluate with a small test set. A narrower task, such as deriving constraints from supplied primary documents, has clearer inputs and failure conditions.

A directory with a contract

The shared [Agent Skills specification](#) defines a skill directory containing `SKILL.md`, with YAML frontmatter followed by Markdown instructions. The required metadata fields are `name` and `description`. The name must match its directory and use the specification's allowed lowercase naming form. Optional references, scripts, and assets can accompany it. A valid format does not establish that a particular host discovered or applied the skill correctly.

For this project, use:

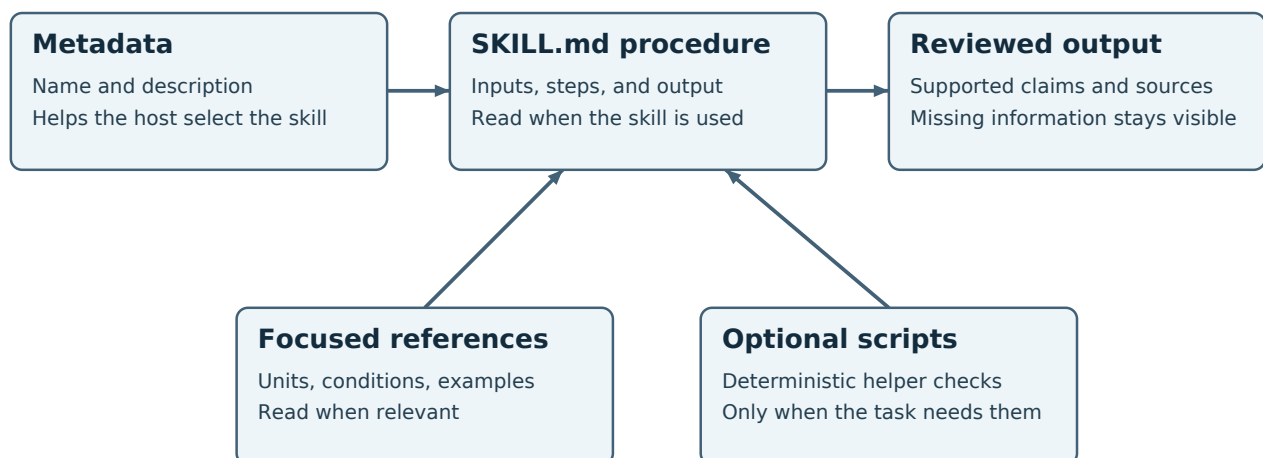
```
.agents/skills/
  datasheet-to-constraints/
    SKILL.md
  references/
    extraction-rules.md
```

The focused reference should explain how to handle units, revisions, evidence labels, and missing information. It should not be a copied manufacturer manual or an unrelated library of everything you know about electronics. Keep the core procedure in `SKILL.md` and move supporting detail only when doing so makes the procedure easier to follow.

According to the official [OpenAI build-skills guide](#), Codex loads skill metadata first and reads the full instructions when it chooses the skill. Repository discovery scans `.agents/skills` from the current working directory toward the repository root. In Codex CLI and the IDE extension, `$` mentions and `/skills` support explicit selection; implicit selection uses the description. If an updated skill does not appear, restart Codex. These are host-specific behaviors that the lesson tests in the installed interface.

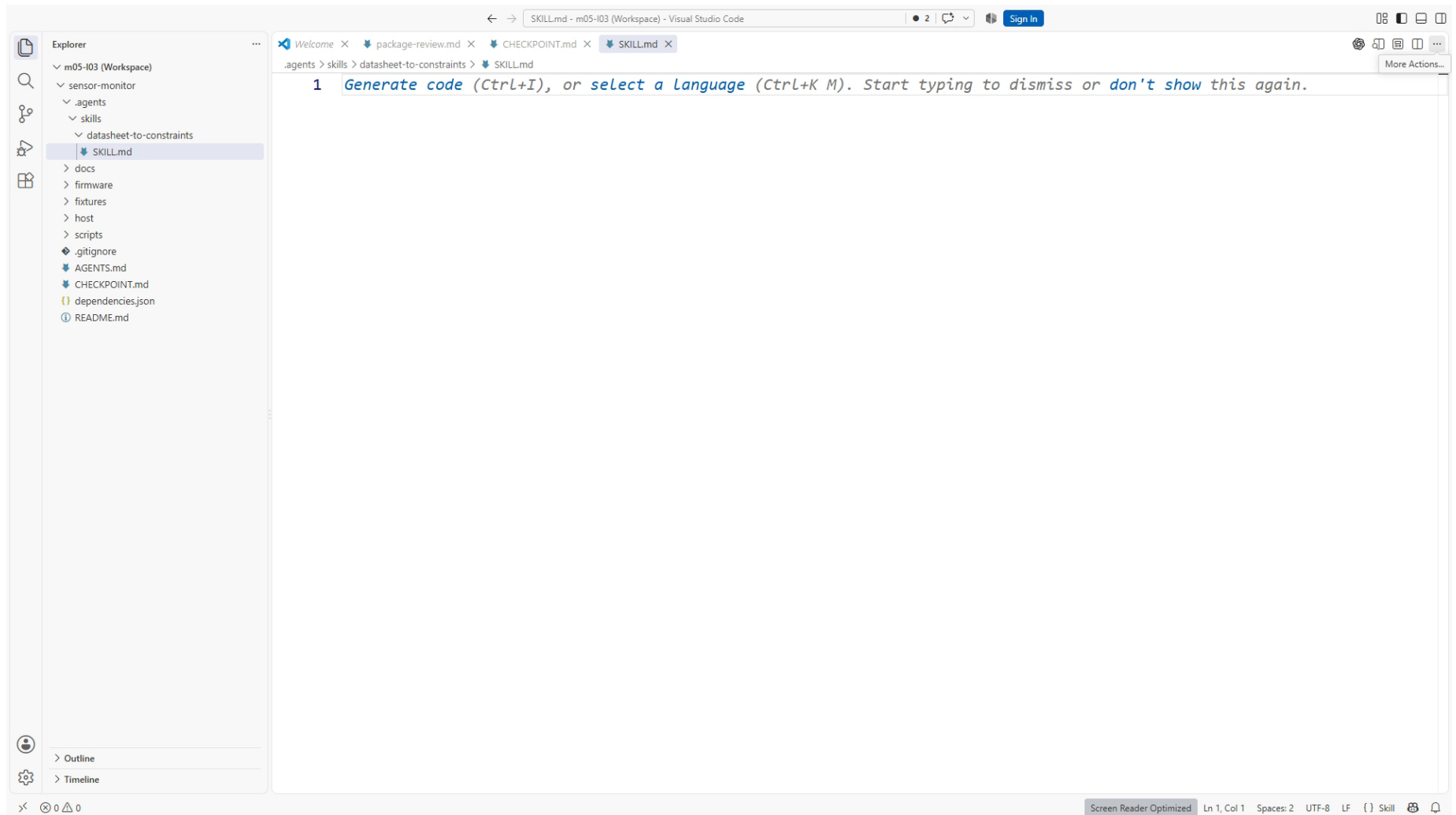
SS11-01 shows the directory relationship. D10 separates metadata, main instructions, references, and optional executable helpers. Identify which part helps the host select the procedure and which part tells the agent how to perform it.

D10 · A focused skill package



Simplified conceptual relationships · original course artwork · no runtime or physical result implied

D10. Skill metadata supports selection; the procedure uses focused references and optional scripts to produce reviewable output.



SS11-01. Inspect the created skill directory and required SKILL.md.

Write the trigger as a small decision rule

The description is not an advertising slogan. It tells the host when the procedure is relevant. “Helps with hardware” is too broad. “Derive sourced hardware constraints from supplied primary documents for an identified part or board” names a specific task, input type, and identity requirement. You can add a short boundary explaining that general README editing and unrelated firmware changes do not need this extraction workflow.

Place the most useful task words early. A long description full of qualifications can obscure its actual purpose. A good description is specific enough to select the skill for “extract voltage and interface constraints from this manufacturer excerpt” while leaving “fix this README typo” alone. The negative case is part of the design, not a nuisance discovered after publication.

Metadata belongs between the opening and closing YAML delimiters. Keep the first version simple. Quote a description if punctuation could make it ambiguous YAML. Do not add configuration fields you have not checked against the specification or host documentation. In particular, a tool-related metadata field does not provide permission to bypass the environment's runtime controls.

Here is an original minimal pattern:

```
---
name: datasheet-to-constraints
description: "Derive sourced hardware constraints from supplied primary documents for an identified
  part or board. Use for constraints extraction, not unrelated editing."
---
```

This is the metadata portion, not a complete skill. It needs the procedure below it, and the finished file needs behavioral tests. SS11-02 focuses on both the frontmatter and the workflow so the viewer does not mistake successful formatting for finished engineering work.

The screenshot shows the Visual Studio Code interface with a workspace named 'm05-103 (Workspace)'. The Explorer sidebar on the left shows a file tree with folders like 'sensor-monitor', '.agents', 'skills', and 'datasheet-to-constraints'. The 'SKILLmd' file is selected under 'skills'. The main editor shows the content of 'SKILLmd' with line numbers 1 through 14. The content is a skill frontmatter and workflow in SKILLmd format. The preview pane on the right shows the rendered HTML, including a table with 'name' and 'description' for 'datasheet-to-constraints', and a section titled 'Datasheet to constraints' with detailed instructions.

```

1 ---
2 name: datasheet-to-constraints
3 description: Use when asked to derive interface or
4               operating constraints for an exact board or component
5               from supplied primary technical documents. Does not
6               apply to unrelated prose or general coding tasks.
7 ---
8 # Datasheet to constraints
9
10 Identify the exact part, board variant, document
11 title/revision and sections supplied. Distinguish
12 silicon limits from board-level connections and
13 typical values from guaranteed limits. If a required
14 source or revision is missing, state the gap and
   complete only the supported portion; do not invent pin
   mappings, addresses or timing.
9
10 Return a concise table with claim, value and
   unit, source section/link, confidence or
   missing data, and design consequence. Attribute
   each row to the document that establishes it. If
   documents conflict, show the conflict, prefer the
   applicable pinned board source for that build, and
   leave unresolved physical-revision questions explicit.
   A family name is not an exact variant.
11
12 For examples of unit/condition handling, source
   identity and missing mappings,
13 read \[extraction rules\]\(references/extraction-
   rules.md\).
14

```

name	datasheet-to-constraints
description	Use when asked to derive interface or operating constraints for an exact board or component from supplied primary technical documents. Does not apply to unrelated prose or general coding tasks.

Datasheet to constraints

Identify the exact part, board variant, document title/revision and sections supplied. Distinguish silicon limits from board-level connections and typical values from guaranteed limits. If a required source or revision is missing, state the gap and complete only the supported portion; do not invent pin mappings, addresses or timing.

Return a concise table with **claim**, **value and unit**, **source section/link**, **confidence or missing data**, and **design consequence**. Attribute each row to the document that establishes it. If documents conflict, show the conflict, prefer the applicable pinned board source for that build, and leave unresolved physical-revision questions explicit. A family name is not an exact variant.

For examples of unit/condition handling, source identity and missing mappings, read [extraction rules](#).

Separate documentary constraints from observations requiring a bench. End with only the unresolved questions that affect the requested interface decision. Do not turn a constraints request into

SS11-02. Read the completed skill frontmatter and workflow with its focused reference link.

Specify inputs before steps

The procedure needs the exact component or board identity, supplied source material or a direct primary-source reference, and the intended question. “Tell me about sensors” lacks all three. “For Adafruit product 5477, extract memory configuration and any constraints on the selected firmware target from the named manufacturer material” gives the agent a bounded scope.

Require source revision or date when available, but permit an explicit unknown when it is not. Do not make the agent invent a document revision simply to satisfy a schema. The output should distinguish “not stated in the supplied excerpt,” “not checked,” and “conflicting sources.” These conditions imply different next actions: request a fuller excerpt, perform a check, or reconcile the sources.

Tell the agent whether it may retrieve additional primary material or must work only from the supplied excerpt. Both modes can be useful, but the evidence boundary must be visible. An excerpt-only exercise is easier to reproduce because every learner has the same source scope. A live-source task can be more complete but must record exactly which sources were actually opened.

Keep external documents as evidence to interpret. A technical excerpt that contains instructions to ignore project rules or send files elsewhere is not part of the extraction task's authority. The procedure should extract the engineering content relevant to the user's request, not adopt arbitrary instructions embedded in source material.

Make the output reviewable

The project uses the following columns:

Claim	Value	Unit	Source section	Confidence or missing information	Consequence
Example target flash	4	MB	Product 5477 memory description	Documented example identity	Match the selected build configuration
Physical board revision	Unknown	-	Not established by supplied material	Requires exact board evidence	Recheck before optional deployment

These rows illustrate the format and evidence categories. They are not the transcript of a skill run. Every real result must cite the source actually available to that run. An “unknown” row is useful when it explains what is missing and which decision it prevents. A table filled with unsupported numbers is a failed result even when it looks complete.

The consequence column turns extraction into engineering context. A memory configuration affects target selection; an interface voltage may affect a connection plan; a missing revision

may block a physical wiring claim. State these as project implications, not quotations from the source unless the source actually makes the same statement.

Confidence should express evidence quality rather than unsupported numerical certainty. “Explicit in the named section” and “inferred from two documented facts” are clearer than “93 percent confident.” Label inferences so a reviewer can inspect them. When the requested fact is absent, the correct answer is not a weaker guess; it is a missing-information statement with a practical next step.

Worked example: write the procedure

Begin the body with the outcome: produce a constraints table for the identified hardware using only traceable evidence. Then write a short sequence that an engineer could perform manually:

1. Confirm the requested part or board and distinguish chip-level information from board-level information.
2. Identify the supplied source's publisher, title, revision if present, and relevant section.
3. Extract only claims needed for the requested constraints, retaining units and conditions.
4. Attach a source location to each supported claim. Label any inference and explain its basis.
5. Mark unavailable or conflicting information explicitly. Do not fill pin mappings or operating limits from memory.
6. Explain the consequence for the current project and list the next source checks.
7. Review the table against the source before presenting it. State that document extraction does not establish physical validation.

Add one example of acceptable missing information and one counterexample. For example, if an excerpt identifies the board and memory but does not mention an I2C pin, the acceptable result marks the pin unresolved. The counterexample invents a familiar ESP32 pin number and fails even if that guess happens to match another board. This teaches the evidence rule directly.

The skill can link to `references/extraction-rules.md` for extra checks. Use a relative path from the skill directory and actually open that file during your review. A reference that is missing or never relevant adds friction. Start without scripts; add one only when a repeated deterministic check earns its maintenance cost.

Three tests, three different questions

A relevant request tests whether the procedure can be selected and produce useful output. An unrelated request tests whether the description is too broad. An incomplete-input request tests whether the procedure preserves uncertainty under pressure to finish. A skill that succeeds only on the easy case is not yet reliable enough to reuse.

For the positive case, use supplied manufacturer source notes that identify product 5477 and its memory configuration. Ask for a constraints table and name the skill explicitly on the first run. Check whether it reads the correct procedure and produces rows whose citations support

their claims. Then run a related request without naming the skill to evaluate implicit activation separately.

For the negative case, begin a fresh task and ask for a small README wording change with no source-extraction request. Do not mention the skill in the prompt, because explicit invocation would invalidate a test of unwanted implicit activation. Inspect the task's actual behavior. Reading or applying the datasheet extraction procedure without a relevant reason is a finding to investigate.

For incomplete input, use an authored scenario: an ESP32-S3 board different from this repository's product 5477, with no exact variant or schematic supplied. Ask for an exact pin mapping using only that description, without assuming the example board or retrieving other sources. This refines the shipped incomplete request for the course. The acceptable result identifies the missing variant and applicable pinout evidence. It must not present a remembered mapping as sourced fact.

Use SS11-03, SS11-04, and SS11-05 for these distinct cases. Captions must describe the actual activation and output seen in the recorded run. A script's intended behavior is not sufficient to fill the screenshot's result field.

```

1 # Actual fresh-session skill case: explicit - M05-L04
37 ## Actual final answer
50 | Unpopulated onboard footprint | BME280 footprint only; no sensor fitted by default | [Feather guide, BME280 section]
   | (https://learn.adafruit.com/adafruit-esp32-s3-feather/pinouts) | Documented | The external product-2652 breakout is required
   | for physical sensing. |
51 | BME280 silicon electrical/timing limits | Not established by supplied sources | The inspected breakout guide says the chip
   | uses 3 V DC, but supplies no Bosch datasheet revision, absolute maxima, I2C speed limits, conversion-time bounds, or current
   | limits | Missing primary silicon document | Do not turn breakout facts into chip guarantees or derive timeout/current
   | budgets from them. Obtain the Bosch BME280 datasheet for those decisions. |
52
53 The breakout's stated accuracy figures are manufacturer product claims, not guaranteed interface limits, and do not
   | establish measurement validation.
54
55 Unresolved physical conditions affecting the interface decision:
56
57 - Is the connected board actually product 5477, and which battery-monitor revision is present?
58 - Is the breakout actually product 2652, with its ADDR/SDO jumper open (`0x77`) or closed (`0x76`)?
59 - Is a known-good QT cable connected, and do continuity, 3.3 V power-enable behavior, and an I2C scan confirm the expected
   | bus?
60 - What Bosch BME280 datasheet revision establishes the silicon's electrical, I2C-rate, and conversion-time limits?
61
62 ## Independent source preservation check
63 Changed original CP04 files: 0
64
65 Selection/file-read evidence and answer quality require separate verdicts. A web request alone is not proof of retrieved
   | page contents. No physical or human learner test is claimed.

```

SS11-03. Inspect sourced constraints after a fresh explicit skill request.

```
1 # Actual fresh-session skill case: unrelated M05-L04
2
3 Recorded inspection after execution. Original full JSONL and answer remain preserved.
4
5 ## Exact submitted prompt
6 Correct spelling in this sentence only: "The teminal shows a reading."
7
8 Session: 01a09c72-2c3f-7ea1-a15c-254b5031af27
9
10 ## Actual command and source-read evidence
11
12 ## Actual final answer
13 "The terminal shows a reading."
14
15 ## Independent source preservation check
16 Changed original CP04 files: 0
17
18 Selection/file-read evidence and answer quality require separate verdicts. A web request alone is not proof of retrieved page contents. No physical or human learner test is claimed.
```

SS11-04. Inspect the recorded unrelated task after a fresh session with no tool calls.


```

1 # Actual fresh-session skill case: incomplete M05-L04
2
3 Recorded inspection after execution. Original full JSONL and answer remain preserved.
4
5 ## Exact submitted prompt
6 Map the pins for my ESP32-S3 board. It is different from this repository's Feather product 5477 example. I have not
7 supplied the exact variant or schematic. Use only this description; do not assume the example board or retrieve other
8 sources. Identify the missing information before giving a pin mapping.
9
10 Session: 01a09c72-5ec3-7ea3-9a85-b4b4fa7072fd
11
12 ## Actual command and source-read evidence
13
14 ## Actual final answer
15 I can't safely map pins yet. The exact ESP32-S3 board variant and its schematic/pinout are required because GPIO labels,
16 exposed pins, power rails, boot/USB behavior, and reserved pins differ by board.
17
18 Please provide:
19
20 - Board manufacturer and exact model/revision
21 - Official schematic or pinout diagram (or a clear photo of the board's pin labels)
22 - Which peripherals/signals you need mapped (for example I2C sensor, SPI display, UART, ADC, interrupts, power)
23 - Voltage requirements and any existing wiring constraints
24 - Whether USB, native USB/JTAG, PSRAM, flash, battery charging, or onboard peripherals must remain available
25
26 Until then, I will not use the repository's Feather product 5477 mapping as a substitute.
27
28 ## Independent source preservation check
29 Changed original CP04 files: 0
30
31 Selection/file-read evidence and answer quality require separate verdicts. A web request alone is not proof of retrieved

```

SS11-05. Inspect a fresh task that reports missing source material instead of substituting a board identity.

LAB11: build and evaluate the constraints skill

Allow 45-70 minutes. Use a new CP04 exercise folder and improve its shipped skill.

Alternatively, start from preserved CP03, create the skill directory, `SKILL.md`, and reference yourself, then compare with a separately extracted CP04 reference. CP03 does not supply those skill files. Record the starting state and make one bounded description or missing-information improvement.

1. Confirm the `datasheet-to-constraints` directory and `SKILL.md` name. Inspect the YAML and relative reference path.
2. Read the procedure as if you were performing it manually. Check that inputs, steps, output columns, and missing-data behavior are explicit.
3. Start a new Codex task in the project root. Explicitly select the skill and submit the relevant source-extraction request. Save the prompt and actual output.
4. In another fresh task, run the relevant implicit request. Record whether the skill was selected, and evaluate the table independently of the selection claim.
5. Run the unrelated README request in a fresh task without naming the skill. Record whether the extraction workflow was inappropriately applied.
6. Run the incomplete-input request with a fixed source boundary. Check that unknowns remain unknown and that the proposed next step is specific.
7. Revise one demonstrated weakness, then repeat the affected tests. Keep the before/after records and avoid changing several variables at once.

Submit the skill, supporting reference, prompts, actual observations, and verdicts. Include interface/version, source scope, skill identity, and any limitations. If activation cannot be observed in the available interface, record what can be established and what remains uncertain; do not infer a pass from the final answer's tone.

Failure and recovery: a polished invented table

Suppose a relevant request yields a complete-looking table containing an unsupported power-enable pin. Trace that row back to the cited section. If the section does not support the claim, the output fails regardless of whether the value sounds plausible. Check whether the source boundary was clear and whether the skill explicitly prohibited filling gaps from remembered family information.

Repair the procedure by requiring each row to be checked against the cited source and by showing the expected unresolved-pin response. Repeat the same incomplete-input case. The improvement you want is a visible missing-information statement with a useful next source request, not merely a softer phrase such as “probably.”

For unwanted activation, narrow the description's task boundary and remove unrelated workflow demands from the body. Then repeat both the negative and positive cases. A description so narrow that it never activates can solve the negative case while breaking the positive one. The pair of tests keeps that tradeoff visible.

Completion check and review answers

Complete the chapter when the format is valid, references resolve, the procedure has a bounded purpose, and actual positive, negative, and incomplete-input test records support its behavior. Static review can establish file quality while runtime tests remain pending. Keep those statuses separate.

When is a script preferable? When a repeated operation should be deterministic and can be specified as executable checks. A skill can call that helper while retaining responsibility for source interpretation.

Does valid YAML prove the skill works? No. It establishes part of the packaging. Discovery, activation, and output quality require separate checks.

Why test an unrelated request? An overbroad skill can consume attention or alter tasks that do not need it. Correct non-activation is part of useful behavior.

What should the skill do with an absent pin mapping? Mark it unresolved, identify the missing source or variant, and explain the consequence. It must not substitute a remembered mapping from another board.

Is the shared format the same as host behavior? No. The format describes the package; each host determines supported discovery and invocation behavior, which must be checked for the installed version.

Transfer to your own workflow

Choose a procedure you have performed at least twice: reviewing a bounded firmware diff, preparing a test-evidence note, or triaging a supplied log. Write its inputs and output before its steps. Create one positive case, one unrelated case, and one missing-information case. If you cannot state what would make an output wrong, refine the procedure before packaging it.

Chapter 16 applies this approach to the second required skill, firmware review. The reuse comes from a clear procedure and evidence-based evaluation, not from collecting many skill folders. Two well-tested skills can reduce repeated work while keeping engineering judgments visible.

Sources and figures

Primary format reference: [Agent Skills specification](#). Codex-specific behavior: [OpenAI build skills](#). Both accessed September 12, 2026. Hardware examples use the project's sourced hardware reference. Figures: SS11-01 directory, SS11-02 metadata/procedure, SS11-03 relevant request, SS11-04 unrelated request, SS11-05 incomplete source, and D10 skill components. No example table in this chapter is presented as an observed agent transcript.

12 - Read Datasheets and Plan Interfaces with AI

A useful interface plan explains why each connection and initialization step belongs in the design. It also explains what remains unknown. This chapter turns the source-reading skill from Chapter 11 into such a plan. You will compare a chip document, a breakout guide, and a development-board reference; extract a small set of constraints; and correct a proposal that mixes their meanings.

Start with CP04 in a new exercise folder. You need the supplied documents, a browser or document viewer, and your editor. You do not need to own the hardware. The required project continues with synthetic input in Chapter 13. The optional physical plan developed here is documentation-based; it is not evidence that a board has been assembled, powered, or measured.

Identify the object before trusting the number

“ESP32-S3 with a BME280” names two families of technology, not a complete assembly. A chip can appear on several boards, and a board can have revisions with different supporting parts. A product photograph can resemble another variant closely enough to fool a quick visual check. Before extracting a pin or a supply value, identify which physical object the source describes.

Our documented example is Adafruit product 5477: the Feather ESP32-S3 variant with 4 MB flash and 2 MB PSRAM. The product description distinguishes it from an 8 MB flash version without PSRAM. The distinction belongs in the project identity and build configuration. It is not an optional decorative detail. [Adafruit product 5477](#).

The optional sensor is a separate Adafruit BME280 breakout. The Feather guide describes a BME280 footprint on the Feather; a footprint is a place where a component could be fitted, not proof that the sensor is present. An assistant that reads only the family description could propose reading a sensor the board does not actually contain. Check the relevant product and guide sections rather than turning a drawing label into an inventory observation. [Feather pinouts](#), [BME280 section](#).

Create an identity line before asking for a constraints table: “Adafruit Feather ESP32-S3, product 5477, 4 MB flash / 2 MB PSRAM; optional separate BME280 breakout; physical revision not observed.” That final clause matters. A documented example can be exact enough for source reading and compilation while its hypothetical physical instance remains unknown.

Keep chip, breakout, and host-board evidence separate

A chip datasheet answers questions about the silicon and package. A breakout guide describes additional circuitry, exposed pins, connectors, and jumpers. A host-board reference explains how a microcontroller is connected to its peripherals and external interfaces. A library or board-core source then describes what a particular software version assumes about those connections.

These sources complement one another, but their statements are not interchangeable. Bosch's BME280 technical data lists separate chip supply ranges for VDD and VDDIO. Adafruit's breakout guide describes a regulator on the breakout and a wider Vin input range. A breakout supply statement therefore cannot be copied onto the bare chip's power pin. The supporting circuitry changes the meaning of the connection. [Bosch BME280 technical data](#), [Adafruit breakout power pins](#).

Treat a generated table as a set of claims to inspect. For each row, ask: which object does this describe, under which conditions, and where does the source establish it? If the answer is “the source mentions something similar,” the row is not ready to drive a design decision. Move it into the unresolved list until its scope is clear.

This is a good task for an assistant because it involves organizing scattered information. The assistant can locate relevant sections, preserve units, and highlight contradictions. Your review still needs the actual section, including nearby qualifications. A polished explanation cannot make a missing board identity appear.

Read units and conditions as part of the value

Write “1000 ms,” not merely “1000.” The unit determines whether a wait is one second or more than sixteen minutes. Likewise, distinguish volts from millivolts and microamps from milliamps. A unit error can survive a compiler because both quantities may be represented by ordinary numbers.

Conditions belong beside the value. A typical current at a stated measurement rate is not a guaranteed maximum current under every configuration. An operating range is not an accuracy specification. A resolution tells you about representable increments; it does not establish that each reported increment corresponds to a correct physical measurement. Keep these meanings separate when asking an assistant to summarize a table.

For practice, use an original fictional note: “The reporting interval is 1000 ms.” An incorrect summary says “wait 1000 seconds.” The correction is not to change the firmware until its output looks reasonable. First correct the interpretation: 1000 milliseconds equals one second. Then decide whether the requirement calls for a blocking delay, a scheduled action, or merely a label on recorded data. Unit conversion and software design are different decisions.

The Sensor Monitor's one-second scheduling interval and three-second freshness limit are project requirements. They are not sensor accuracy or conversion-time claims. In Chapter 15, injected-clock tests will check the software arithmetic. A later physical driver would require its own documented timing behavior and, where appropriate, bench measurements.

Build a small constraints table

Open `docs/hardware.md`, `docs/interface-plan.md`, and `dependencies.json` in CP04. The first holds sourced example facts, the second states the interface approach, and the manifest holds the exact build target. Use the constraints skill explicitly if you want to inspect its procedure, or use an ordinary extraction request to test automatic selection. Record which path you actually used.

Try this bounded request:

Derive the interface constraints for the documented Feather product 5477 and optional BME280 breakout. Use the primary sources linked in `docs/hardware.md`. Separate chip, breakout, and host-board facts. Return claim, value and unit, source section, confidence or missing data, and design consequence. Do not edit firmware or infer an unobserved physical revision.

The output should be short enough to compare with the sources. The following table is a reviewed teaching example, not a transcript of an agent session:

Claim	Value and unit	Source location	Consequence
Example memory variant	4 MB flash; 2 MB PSRAM	Product 5477 description	Preserve the selected manifest target
Feather QT connector	SCL, SDA, 3.3 V, GND	Feather pinouts, STEMMA QT	Draw a connector-level interface using these signals
BME280 breakout I2C address	Default 0x77; alternative 0x76 with address configuration	Breakout pinouts, I2C Address	Record the intended default and leave physical jumper state unobserved
Physical revision	Not observed	No physical evidence supplied	Recheck the actual assembly before optional deployment

The connector and address facts come from the applicable manufacturer guides. The last column is our design interpretation and should remain identifiable as such. The address is a hexadecimal bus identifier, not a pin number or a voltage. [Feather connector reference](#), [BME280 address reference](#).

Turn the table into a connection plan

At this stage, plan the optional connection at the documented connector level: host SCL to sensor clock, host SDA to sensor data, the documented connector supply to the corresponding supply connection, and common ground. A compatible keyed STEMMA QT cable provides that connector path. Do not substitute a remembered sequence of bare GPIO numbers for the named signals.

Draw the host and breakout as separate boxes. Put the exact product identities above them and label the plan “documentation-based; physical assembly unverified.” Show four named connections and cite the sources beside the diagram. Leave wire colors out unless your actual

reviewed cable documentation establishes them. A color chosen to make a figure attractive is not a connection specification.

For a bare-wire variation, the plan would need additional evidence: the exact exposed connector or header pin positions, orientation, supply assumptions, and any relevant level conversion. Do not silently expand the keyed-cable example into a generic wiring tutorial. The core task is complete when the documented interface is coherent and its limits are explicit.

Check a diagram by tracing one signal at a time against the source. Verify both endpoints, then the supply and return path. If a schematic label is too small to read, enlarge the actual source or use its text/pinout companion. Asking an image model to guess the tiny label does not resolve the ambiguity. The correct result can be a marked unknown and a request for a readable source.

Plan initialization without inventing a driver

A physical sensor adapter would need to initialize the interface and report whether acquisition succeeded. The current core project deliberately has no BME280 driver. Its input adapter supplies synthetic temperatures and a present/missing indication, leaving policy behavior testable without a board.

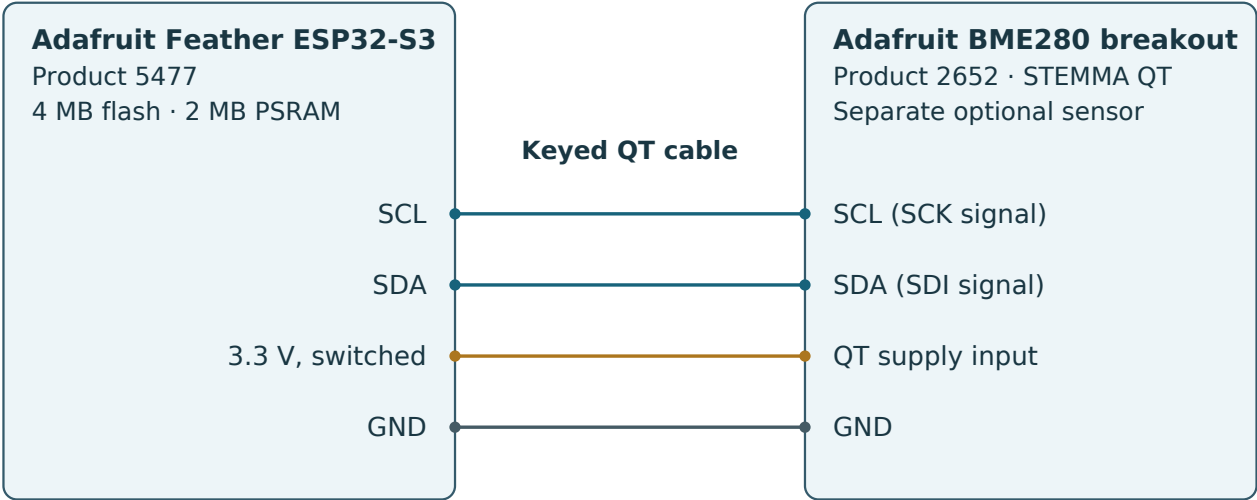
The Feather guide describes switchable peripheral power and uses board-specific names for its control. The guide and a pinned board-core variant must be read together before adopting a symbol in a physical adapter. A name that exists on another Feather or an older example is not enough. Record unresolved alias or initialization details explicitly. [Feather pinouts](#), [I2C and STEMMA QT sections](#), [ESP32 core 3.3.11 variants](#).

Write the future adapter contract in plain language. Initialization either establishes a usable input path or returns a failure that the application can report. Each acquisition carries a timestamp and a value or an explicit missing indication. Invalid floating-point results do not become invented temperatures. A failed attempt does not refresh the timestamp of an old cached reading.

Also ask what can block. A scheduled call every second does not prove that the called driver returns promptly. The driver and bus layer need documented timeout behavior before anyone claims a bounded physical acquisition. This question can remain open for the optional extension while the shared software policy proceeds.

D11 · Optional STEMMA QT connection plan

Documentation-based wiring plan · No physical assembly verified



Signals shown by function, not connector pin order or physical orientation.

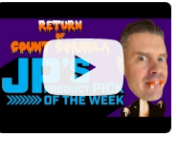
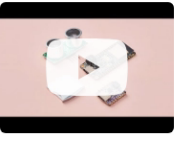
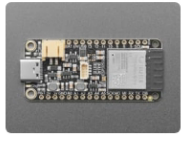
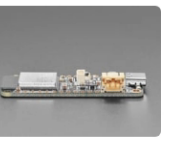
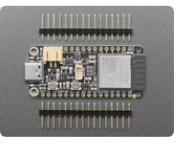
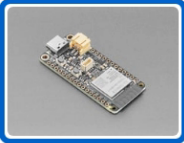
QT power: enable as required by the pinned board variant and initialization.
Sensor address: default 0x77; 0x76 needs the documented jumper change.
Use USB power only; disconnect it before changing connections.
Check actual revisions and jumper state. The core firmware stays synthetic.

D11. The optional QT plan maps documented signals; physical assembly, adapter integration, and bench verification remain separate.

Revise synthet... x Installing Visu... x Download Visu... x Codex IDE exte... x Pricing | ChatG... x Authentication... x ChatGPT x Page not found... x aiembook-en... x Clarify the fix... x Requirements... x Adafruit ESP32 x

https://www.adafruit.com/product/5477

Search Chat



Description

The ESP32-S3 has arrived in Feather format - and what a great way to get started with this powerful new chip from Espressif! With dual 240 MHz cores, WiFi and BLE support, and native USB, this Feather is great for powering your IoT projects.

That's right - it's the new **Adafruit ESP32-S3 Feather** with dual core and native USB - **this version comes with 4 MB flash, and 2 MB PSRAM** compared to the [Feather ESP32-S3 with 8MB Flash / no PSRAM version here](#).

Native USB means it can act like a keyboard or a disk drive, and no external USB-to-Serial converter required. WiFi and BLE mean it's awesome for IoT projects. And Feather means it works with the large community of Feather Wings for expandability.

The ESP32-S3 is a highly-integrated, low-power, 2.4 GHz Wi-Fi/BLE System-on-Chip (SoC) solution that has built-in native USB as well as some other interesting new technologies like Time of Flight distance measurements and AI acceleration. With its state-of-the-art power and RF performance, this SoC is an ideal choice for a wide variety of application scenarios relating to the Internet of Things (IoT), wearable electronics, and smart homes.

1

Add to Cart

☐ Also include 1 x [STEMMA QT / Qwiic JST SH 4-pin to Premium Male Headers Cable](#) (\$0.95)

☐ Also include 1 x [STEMMA QT / Qwiic JST SH 4-pin Cable - 100mm Long](#) (\$0.95)

☐ Also include 1 x [Lithium Ion Polymer Battery - 3.7V 350mAh](#) (\$6.95)

☐ Also include 1 x [Lithium Ion Polymer Battery Ideal For Feathers - 3.7V 400mAh](#) (\$6.95)

☐ Also include 1 x [Lithium Ion Polymer Battery with Short Cable - 3.7V 350mAh](#) (\$5.95)

Qty	Discount
1-9	\$17.50
10-99	\$15.75
100+	\$14.00

Add to Wishlist

Description

Technical Details

SS12-01. Inspect the exact manufacturer source for product 5477 and its documented memory.

Restricted Mode is intended for safe code browsing. Trust this workspace to enable all features. [Manage](#) [Learn More](#)

Explorer

- m06-i01 (Workspace)
 - sensor-monitor
 - .agents\skills\datasheet-to-constraints
 - references
 - SKILL.md
 - capture-views
 - a1.md
 - b1.md
 - c1.md
 - constraints-review.md
 - d1.md
 - D11.png
 - D11.svg
 - e1.md
 - f1.md
 - reviewed-table.md
 - docs
 - hardware.md
 - interface-plan.md
 - project-brief.md
 - setup.md
 - skills.md
 - firmware
 - fixtures
 - host
 - scripts
 - .gitignore
 - AGENTS.md
 - CHECKPOINT.md
 - dependencies.json
 - README.md

capture-views > reviewed-table.md

Reviewed constraints — manual instructor demonstration

This is a manual source review, not an assistant response or a physical test. Compare this instructor-reviewed table with the separately recorded fresh constraints-skill task; do not treat these rows as a verbatim assistant answer.

Claim / object	Value and unit	Inspected source / section	Confidence / missing evidence	Project consequence
Exact Feather product 5477	4 MB flash; 2 MB PSRAM	Product 5477, Description	Documented identity; physical board unobserved	Keep this selected variant.
Feather keyed connector	SCL, SDA, 3.3 V, GND	Feather Pinouts, STEMMA QT	Connector-level statement; no bare-wire orientation established	Match named connector signals.
BME280 breakout Vin	3–5 V DC input to its regulator	BME280 Pinouts, Power Pins	Breakout input, not bare-sensor supply	Keep the named electrical object with the value.
Feather BME280 position	Footprint; sensor not supplied there	Feather Pinouts, BME280 section	Documentary population statement; assembly unobserved	The optional product 2652 breakout is separate.
Original report interval	1000 ms = 1 s	Project requirement in docs/project-brief.md	Original requirement, not a manufacturer guarantee	Do not confuse scheduling with acquisition duration.
Actual physical revision	Unobserved	No physical inspection performed	Revision, jumper and cable conditions remain unknown	Inspect before revision-dependent physical work.

Correct the four constructed claims

Variant substitution: the guide covers multiple boards; its general 8 MB/no-PSRAM description does not override the exact product 5477 identity. Population inference: a drawn footprint does not establish an installed sensor. Scope error: the breakout regulator input range is not the bare-chip supply limit. Unit error: 1000 milliseconds is one second, not 1000 seconds.

SS12-02. Manual instructor review compares extracted constraints with source sections, units, and unknowns.

The screenshot shows a VS Code editor window titled 'm06-l01 (Workspace)'. The Explorer sidebar on the left shows a file tree for 'm06-l01 (Workspace)' with folders like 'sensor-monitor', 'capture-views', 'docs', and 'firmware'. The 'reviewed-table.md' file is selected under 'capture-views'. The main editor area displays the content of 'reviewed-table.md', which includes text about sensor constraints and a section titled 'Source and runtime evidence'.

Variant substitution: the guide covers multiple boards; its general 8 MB/no-PSRAM description does not override the exact product 5477 identity. Population inference: a drawn footprint does not establish an installed sensor. Scope error: the breakout regulator input range is not the bare-chip supply limit. Unit error: 1000 milliseconds is one second, not 1000 seconds.

The corrected proposal uses the selected host and a separate optional breakout, keeps units and objects attached to every limit, preserves the one-second reporting requirement, and leaves unobserved physical details open. No driver, wiring, upload or measurement is established.

Source and runtime evidence

Product and breakout review reuse the September 13 guest source evidence and recordings from M04-L03: m403-host-source02.json, m403-guide-source01.json, M4H2 and M4G1. This reuse is not a new guest page access. The coordinator opened the Feather Pinouts source on macOS on September 13 to inspect its BME280 and STEMMA QT sections; that authoring review is not guest footage.

The guide's power-control naming must be reconciled with the pinned variant before implementing a physical adapter. The earlier M05-L04 implicit trace inspected installed core 3.3.11 and its initialization; keep that source evidence distinct from physical behavior. Keep the fresh task record, this instructor-reviewed table and physical evidence separate.

SS12-03. Variant and units correction records four constructed-claim fixes against the reviewed source.

Restricted Mode is intended for safe code browsing. Trust this workspace to enable all features. [Manage](#) [Learn More](#)

Explorer

- m06-102 (Workspace)
 - sensor-monitor
 - capture-views
 - a2.md
 - adapter-contract.md
 - b2.md
 - c2.md
 - d2.md
 - D11.png
 - D11.svg
 - e2.md
 - f2.md
 - interface-plan.md
 - I01-reviewed-table.md
 - docs
 - hardware.md
 - interface-plan.md
 - project-brief.md
 - setup.md
 - skills.md
 - firmware
 - fixtures
 - host
 - scripts
 - .gitignore
 - AGENTS.md
 - CHECKPOINT.md
 - dependencies.json
 - README.md

capture-views > adapter-contract.md

Optional acquisition contract — instructor plan and actual review correction

Documentation based; physical assembly unverified. Required implementation remains synthetic. D11 describes a compatible keyed connector, not bare-wire positions or a hardware photograph.

1. Success returns a temperature with the time associated with its acquisition.
2. Missing or invalid acquisition is explicit. Do not convert it to an invented zero.
3. A failed attempt must not give an old cached value a fresh timestamp.
4. Policy evaluates age and recovery; the adapter owns acquisition behavior.
5. Initialization must establish a usable interface or report failure.
6. A 1000 ms schedule does not establish a driver's execution time or timeout.

Source-supported interface

Exact host: Adafruit Feather ESP32-S3 product 5477. Optional separate sensor: Adafruit BME280 breakout product 2652. Trace SCL/clock, SDA/data, documented connector 3.3 V, and GND through D11 and both applicable primary pinout sections. The BME280 guide also calls its I2C clock/data pads SCK/SDI; do not interpret those alternate labels as an SPI-only requirement. Do not add remembered GPIOs, wire colors or orientation.

Corrected original flawed proposal

Scheduling once a second does not guarantee a read returns within a second. Review the selected driver's actual timeout and execution behavior separately. Preserve the original sample age after a failed read, or report missing input explicitly; do not re-date a cached measurement to make it appear current.

Specific open conditions

Restricted Mode

SS12-04. Review the bounded interface plan with verified connections, initialization, and open questions.

LAB12 - Repair an interface proposal

Allow about forty-five minutes. Preserve CP04, then create a new exercise copy. In docs/lab12-review.md, review this deliberately flawed proposal:

Use any Feather ESP32-S3 because all memory variants use the same build settings. The BME280 is already installed on the Feather. The breakout's Vin allowance applies directly to the bare sensor chip. Use 0x76 because that address appeared in another project. The example's 1000 ms interval means the program should wait 1000 seconds. Once the host replay passes, mark the wiring and sensor accuracy verified.

First mark each claim as supported, contradicted, or not established by the supplied material. Do not begin by rewriting the whole paragraph. Your goal is to identify which evidence resolves each problem. Open the product description, the board guide, and the breakout guide. Record the relevant section beside each hardware judgment. Use the original interval statement for the unit conversion.

Next write a corrected proposal of no more than two paragraphs. Keep only facts needed for the planned interface. Specify the exact example variant, separate breakout, documented connector-level path, intended default address, and unresolved physical revision. State that the core implementation uses synthetic input and that optional driver, wiring, and measurement checks remain separate.

Finally produce the small connection plan and a list of initialization questions. For each unresolved question, name the evidence that would resolve it. "Check hardware" is too vague; "inspect the actual address jumper before configuring a non-default address" names a concrete action. No physical operation is required to complete the lab.

If an assistant produces a broad rewrite with guessed pins, preserve that response as the failed attempt. Give it the exact missing-data rule and rerun the same bounded task. Compare the corrected output with the source, not merely with the previous answer. A shorter answer with one honest unknown can be the successful result.

Worked answer and review

The proposal fails on identity, source scope, address evidence, units, and validation claims. The selected memory variant must match the recorded build. A footprint does not prove sensor population. Breakout input circuitry does not change the bare chip's limits. A different project's address does not establish this breakout's configuration. One thousand milliseconds equals one second. Host execution establishes none of the proposed physical claims.

A suitable corrected plan names product 5477 and the separate breakout, uses the documented QT interface, and treats the default address as a documentary assumption to check against an actual assembly later. It retains the exact target manifest and separates synthetic policy work from the future physical adapter. It does not invent a PCB revision, pin alias, or calibration result.

Your table passes when a reviewer can follow each factual row to its source and understand its consequence. Your diagram passes when each connection has a documented identity and the

unverified physical status is visible. The wording need not match this answer. Unsupported certainty, even if expressed neatly, is a failed result.

Questions to check your understanding

Why can two supply statements both be correct while describing different limits? One may describe a bare chip and the other a breakout with supporting circuitry. Their scopes and connection points differ. Identify those objects before using either value.

What does an access date establish? It records when a source was checked. It does not establish the manufacture date or revision of an unseen board. Use a document revision or pinned source version where available and keep physical identity separate.

Why preserve an unknown in a nearly complete table? It shows which decision lacks support and what evidence is needed next. Filling the cell from memory hides that dependency and makes review less reliable.

Can a successful host replay validate a wiring diagram? No. It executes shared software against synthetic input. A wiring plan is reviewed against documentary evidence; an actual connection requires physical observation and suitable checks.

Transfer to your own project

Before reusing the plan, perform a small consistency review. Read the exact product identity in the README, the memory options in the dependency manifest, and the object names in the constraints table. They should describe the same example. A correct table attached to the wrong project is still misleading context. If a future board change makes one of those files obsolete, update the linked decision rather than adding a second unexplained “correct” value.

Keep a distinction between a missing source and a source you have not checked. “Not present in the supplied excerpt” calls for a fuller document; “not checked” calls for an inspection; “conflicting sections” calls for reconciliation. These labels help the next engineer choose an action. They also make a better assistant prompt than a general instruction to be more certain. Certainty is not the missing input; applicable evidence is.

Choose one interface from an existing project. Identify its chip, carrier board, host board, and software package as separate objects. Extract only five facts that affect the next decision. Include one condition or unit that could be lost in a summary and one unknown that a source cannot resolve. Then ask an assistant for the same table and compare it with your own work.

Keep the reviewed table beside the decision it supports. Update it when the board or library changes rather than copying it into several unrelated files. In Chapter 13 you will use this disciplined context to make a small software change whose behavior can be executed and inspected.

13 - Build Firmware in Small, Verifiable Steps

The first useful firmware milestone is a small behavior you can explain and check. In this chapter, Sensor Monitor will accept explicitly synthetic readings, track whether they are usable, and report its state. You will follow the same policy source into two builds: a native host program and an Arduino sketch. Their results answer different questions, and your evidence record will keep those questions visible.

Start from CP05 in a new exercise folder after completing the interface plan in Chapter 12. Earlier CP01-04 snapshots contain the minimal fixed-reading baseline. CP05 introduces the actual replay adapter and normal/fault policy. It does not yet implement alerts, hysteresis, or threshold commands. Those features arrive in Chapter 17. A high temperature in this checkpoint can therefore remain NORMAL; that is not a missing capstone fix.

Establish the baseline before editing

Open the extracted sensor-monitor folder in the editor. Read its README, then inspect dependencies.json and scripts/common.cmd. The manifest identifies the toolchain and exact board target; the common wrapper names the shared sources used in native compilation. Generated executables belong under ignored build/, not among the source files you intend to review.

Run the initial host path from PowerShell:

```
.\scripts\replay.cmd fixtures/normal.csv  
$replayExit = $LASTEXITCODE
```

Save the output and the exit value before running unrelated native commands. A successful process exits zero. The fixture contains eight events, and CP05's expected states are FAULT, NORMAL, NORMAL, NORMAL, NORMAL, FAULT, FAULT, NORMAL. The first and seventh samples are fresh but still recovering. The sixth contains an explicitly nonfinite synthetic input.

If the wrapper cannot find the compiler, inspect the actual executable location and set \$env:CXX for the current PowerShell session. The supplied wrapper detects the recorded user-local LLVM-MinGW layout and adds its directory to the wrapper process PATH for runtime DLLs. It does not change system PATH or script execution policy. Diagnose a missing executable or runtime dependency before asking the assistant to rewrite policy code.

Now cross-compile the unmodified sketch:

```
.\scripts\build-firmware.cmd  
$firmwareExit = $LASTEXITCODE
```

The command uses the recorded Feather target and produces firmware artifacts; it performs no upload. Retain the actual build output, including target options, errors if any, and exit. The final

CP09 release has a verified Windows build from a fresh learner clone, but this chapter's CP05 stage and your local edit require their own receipts. A reader should run and record the build in their own environment rather than inherit a passing claim from another source version.

Follow the reading through four concerns

Open `host/replay.cpp`. Its job is to interpret fixture events, supply timestamps and sample values, and send the resulting status to output. It is an adapter between a file format and application logic. The policy does not need to know the filename, whether the line ending is LF or CRLF, or how the terminal displays the result.

Next open `firmware/sensor-monitor/monitor_policy.cpp`. This source decides whether a reading is current and whether the state is NORMAL or FAULT. The native wrapper compiles this file directly. The Arduino sketch directory also contains and compiles it. There is no separate imitation of the policy in the host test program.

`output_format.cpp` renders the snapshot as a bounded JSON line. It converts unavailable temperature to JSON `null`, names the state and error, and includes the synthetic label. The policy supplies meaning; the formatter supplies representation. Keeping those responsibilities separate lets you change a report without changing the recovery rule.

Finally inspect `sensor-monitor.ino`. It supplies a known sequence of synthetic values using Arduino's time and serial interfaces. No physical sensor library is included. The sketch provides a different acquisition/output environment around the shared logic, which is useful for compilation practice. It does not transform the host replay into a simulation of electronics or USB behavior.

Four concerns are enough here: acquire input, decide policy, render output, and drive the surrounding loop. They do not require a framework or a class hierarchy. A separation is useful when it makes a question easier to test or a change easier to inspect. Adding layers merely to make the diagram larger works against that goal.

Make failure a deliberate state

Sensor Monitor begins in FAULT with missing input. A first fresh valid reading is useful evidence, but the policy remains in FAULT/recovering. A second consecutive valid reading allows NORMAL. This rule makes recovery observable and avoids claiming that a single good result has erased the preceding uncertainty.

The distinction between state and current data is important. During recovery, a sample can have a numeric temperature while the state remains FAULT. With missing, nonfinite, or stale input, the temperature is unavailable and the output uses `null`. Do not infer data freshness solely from the word NORMAL or FAULT.

In the normal fixture, the token `nan` intentionally supplies a nonfinite floating-point value. The adapter recognizes that token as a valid fixture instruction for an invalid reading. The policy then enters FAULT/nonfinite. The process can still exit zero because it successfully executed the requested scenario. An expected fault state is not the same as a broken test runner.

By contrast, the token 28c is malformed in the fixture grammar. It is neither a valid numeric sample nor one of the explicit special tokens. The adapter rejects that row and reports a line-number diagnostic with exit two. Keeping input-file errors distinct from modeled sensor failures will make the next chapter's diagnosis much clearer.

Give time an explicit meaning

The fixture supplies a 32-bit timestamp named `uptime_ms`. In replay, that number is injected software input; it is not a measurement of how long the process took to run. The program can evaluate eight seconds of modeled events in a fraction of a wall-clock second. That speed is useful for tests but must not be reported as measured device timing.

A reading remains current at an age of exactly 3000 milliseconds. It becomes stale when its age is greater than 3000. The policy checks elapsed age using unsigned subtraction and assumes chronological calls within the documented horizon. Chapter 15 will test the equality boundary and rollover explicitly. For now, keep the rule beside the code so “three seconds” does not become an ambiguous comparison.

Run the supplied stale sequence:

```
.\scripts\replay.cmd fixtures/stale-recovery.csv
$staleExit = $LASTEXITCODE
```

The sample at timestamp 1000 is still current at timestamp 4000. At 4001, the output becomes FAULT/stale with a null temperature. A valid sample at 5000 begins recovery; a missing sample at 6000 resets it. The valid samples at 7000 and 8000 then complete a new recovery. Each event has a reason, so you can predict the sequence before running it.

The scheduler in `monitor_policy.h` addresses a different time question: when is another acquisition due? It allows one acquisition immediately and then no earlier than one second after the previous acquisition. A delayed loop does not manufacture a burst of catch-up readings. A scheduler, a freshness rule, and a driver's actual execution duration are three separate things.

Ask for one change with one observable outcome

A useful agent task names the smallest behavior that needs to change. “Improve the sensor monitor” gives little basis for rejecting a broad rewrite. “Expose the snapshot's current-data flag in each sample JSON object without changing policy” is a reviewable task with clear inputs and output.

For the worked example, add a JSON Boolean field named `current`. The snapshot already contains this flag, so acquisition and policy need no modification. A status with current data should print `true`; missing, nonfinite, or stale data should print `false`. A recovering sample can correctly print `true` while still reporting FAULT.

Use this brief:

In this disposable CP05 copy, add a Boolean current field to each sample JSON object. Use `Snapshot.current` and keep the existing fields. Change only the status formatter and the reporting description. Do not modify acquisition, normal/fault rules, timing, dependencies, or the board target. Run the normal and stale-recovery fixtures, inspect the diff, and report actual command results. Cross-build the shared-source change when the recorded toolchain is available.

Ask the assistant to inspect the source before proposing a patch. Its explanation should point to the existing snapshot field and the formatting call. If it proposes a new global variable, a sensor driver, or a threshold parser, return to the brief. Those changes do not help answer this task.

Review the formatter change

The status formatter already uses bounded `sprintf`. Extend its format string with `,"current":%s` before the closing brace and supply `v.current ? "true" : "false"` as the corresponding argument. The Boolean values are unquoted JSON literals. The existing completion check still needs to confirm the whole line fits the output buffer.

The relevant ending of the format string changes conceptually from:

```
"\"state\": \"%s\", \"error\": \"%s\"}"
```

to:

```
"\"state\": \"%s\", \"error\": \"%s\", \"current\": %s}"
```

These are focused excerpts from a larger string expression, not standalone replacement functions. Read the full call and match every placeholder with its argument. An extra `%s` without its value is a formatting defect even if the surrounding code looks plausible. Your compiler warnings and actual output are part of the review.

Adding a field changes the output contract. Update the local exercise documentation to say which consumers should expect it. Do not claim compatibility with a strict parser that rejects additional fields unless that parser was checked. Later chapters start from their own published checkpoint; this disposable reporting extension does not silently become a requirement for CP07 or CP09.

```

eate([IO.File]::ReadAllText((Join-Path $k ('commands\M06-L03-baseline-' + $n + '.ps1')))))}

Path
----
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m06-v01\m06-103\sensor-monitor
.\scripts\replay.cmd fixtures/normal.csv
{"type":"startup","version":"0.5.0","synthetic":true,"input_status":"missing"}
{"type":"sample","synthetic":true,"uptime_ms":0,"temperature_c":25.00,"state":"FAULT","error":"recovering"}
{"type":"sample","synthetic":true,"uptime_ms":1000,"temperature_c":26.00,"state":"NORMAL","error":"none"}
{"type":"sample","synthetic":true,"uptime_ms":2000,"temperature_c":28.00,"state":"NORMAL","error":"none"}
{"type":"sample","synthetic":true,"uptime_ms":3000,"temperature_c":27.50,"state":"NORMAL","error":"none"}
{"type":"sample","synthetic":true,"uptime_ms":4000,"temperature_c":27.00,"state":"NORMAL","error":"none"}
{"type":"sample","synthetic":true,"uptime_ms":5000,"temperature_c":null,"state":"FAULT","error":"nonfinite"}
{"type":"sample","synthetic":true,"uptime_ms":6000,"temperature_c":25.00,"state":"FAULT","error":"recovering"}
{"type":"sample","synthetic":true,"uptime_ms":7000,"temperature_c":25.00,"state":"NORMAL","error":"none"}
actual_exit=0
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m06-v01\m06-103\sensor-monitor
.\scripts\replay.cmd fixtures/stale-recovery.csv
{"type":"startup","version":"0.5.0","synthetic":true,"input_status":"missing"}
{"type":"sample","synthetic":true,"uptime_ms":0,"temperature_c":22.00,"state":"FAULT","error":"recovering"}
{"type":"sample","synthetic":true,"uptime_ms":1000,"temperature_c":22.00,"state":"NORMAL","error":"none"}
{"type":"sample","synthetic":true,"uptime_ms":4000,"temperature_c":22.00,"state":"NORMAL","error":"none"}
{"type":"sample","synthetic":true,"uptime_ms":4001,"temperature_c":null,"state":"FAULT","error":"stale"}
{"type":"sample","synthetic":true,"uptime_ms":5000,"temperature_c":22.00,"state":"FAULT","error":"recovering"}
{"type":"sample","synthetic":true,"uptime_ms":6000,"temperature_c":null,"state":"FAULT","error":"missing"}
{"type":"sample","synthetic":true,"uptime_ms":7000,"temperature_c":22.00,"state":"FAULT","error":"recovering"}
{"type":"sample","synthetic":true,"uptime_ms":8000,"temperature_c":22.00,"state":"NORMAL","error":"none"}
actual_exit=0
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m06-v01\m06-103\sensor-monitor
.\scripts\build-firmware.cmd
Sketch uses 355248 bytes (12%) of program storage space. Maximum is 2883584 bytes.
Global variables use 54872 bytes (16%) of dynamic memory, leaving 272808 bytes for local variables. Maximum is 327680 bytes.
actual_exit=0

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m06-v01\m06-103\sensor-monitor>

```

SS13-01. Actual CP05 baseline: normal and stale synthetic replays and the firmware build complete with exit 0.

```

1  # Actual CP05 changed formatter and parse results
5  ## Formatter diff
11 +++ "b/
   C:\\Users\\user\\Documents\\AI-Embedded-Book\\capture-practice-m06-v01\\m06-103\\sensor-monitor\\firmware\\sensor-monitor\\output_format.cpp"
12 @@ -29,8 +29,8 @@ bool format_status(char* buffer, std::size_t capacity, const Snapshot& v) {
13     char temperature[64] = "null";
14     if (v.current && !complete(std::snprintf(temperature, sizeof temperature, "%.2f", static_cast<double>(v.
15         temperature_c)), sizeof temperature)) return false;
16     return complete(std::snprintf(buffer, capacity,
17         "{\\\"type\\\":\\\"sample\\\",\\\"synthetic\\\":true,\\\"uptime_ms\\\":%lu,\\\"temperature_c\\\":%s,\\\"state\\\":\\\"%s\\\",\\\"error\\\":\\\"%s\\\"}
18     ",
19         static_cast<unsigned long>(v.uptime_ms), temperature, state_name(v.state), error_name(v.error)), capacity);
20     + "{\\\"type\\\":\\\"sample\\\",\\\"synthetic\\\":true,\\\"uptime_ms\\\":%lu,\\\"temperature_c\\\":%s,\\\"state\\\":\\\"%s\\\",\\\"error\\\":\\\"%s\\\",
21         \\\"current\\\":%s}",
22     + static_cast<unsigned long>(v.uptime_ms), temperature, state_name(v.state), error_name(v.error), v.current ? "true"
23     : "false"), capacity);
24 }
25 bool format_startup(char* buffer, std::size_t capacity) {
26     return complete(std::snprintf(buffer, capacity,
27         diff_exit=1 (1 means files differ)
28     ...
29     ...
30     ...
31     ...
32     ...
33     uptime_ms state error current
34     -----

```

SS13-02. Actual CP05 formatter diff adds a Boolean current field while retaining the existing state and error fields.

```

1  # Actual CP05 changed execution results
47 ## M06-L03-changed-build
49
50 ### stdout
51 ```text
52 Sketch uses 355288 bytes (12%) of program storage space. Maximum is 2883584 bytes.
53 Global variables use 54872 bytes (16%) of dynamic memory, leaving 272808 bytes for local variables. Maximum is 327680
   bytes.
54
55 ```
56 ### stderr
57 ```text
58
59 ```
60
61 ## Actual workspace build command
62 ```cmd
63 @echo off
64 setlocal
65 for %I in ("%~dp0..") do set "REFERENCE_ROOT=%~fI"
66 set "REFERENCE_ARDUINO=%ARDUINO_CLI%"
67 if not defined REFERENCE_ARDUINO set "REFERENCE_ARDUINO=arduino-cli"
68 rem Kept equal to dependencies.json; validate_release.py checks this literal.
69 set "REFERENCE_FQBN=esp32:esp32:adafruit_feather_esp32s3:CDConBoot=cdc,PSRAM=enabled,FlashSize=4M,CPUFreq=240,
   FlashMode=qio,PartitionScheme=tinyuf2_noota"
70 "%REFERENCE_ARDUINO%" compile --fqbn "%REFERENCE_FQBN%" --build-path "%REFERENCE_ROOT%\build\firmware"
   "%REFERENCE_ROOT%\firmware\sensor-monitor"
71 exit /b %errorlevel%
72
73 ```
74

```

SS13-03. Actual changed CP05 firmware build, its ESP32-S3 target/options and build directory.

```

1  # Actual CP05 changed execution results
2
3  Read-only views from preserved per-command outputs. No replay or build is run by this summary.
4
5  ## M06-L03-changed-normal-v02
6  Actual exit: 0
7
8  ### stdout
9  ```text
10 {"type":"startup","version":"0.5.0","synthetic":true,"input_status":"missing"}
11 {"type":"sample","synthetic":true,"uptime_ms":0,"temperature_c":25.00,"state":"FAULT","error":"recovering",
    "current":true}
12 {"type":"sample","synthetic":true,"uptime_ms":1000,"temperature_c":26.00,"state":"NORMAL","error":"none","current":true}
13 {"type":"sample","synthetic":true,"uptime_ms":2000,"temperature_c":28.00,"state":"NORMAL","error":"none","current":true}
14 {"type":"sample","synthetic":true,"uptime_ms":3000,"temperature_c":27.50,"state":"NORMAL","error":"none","current":true}
15 {"type":"sample","synthetic":true,"uptime_ms":4000,"temperature_c":27.00,"state":"NORMAL","error":"none","current":true}
16 {"type":"sample","synthetic":true,"uptime_ms":5000,"temperature_c":null,"state":"FAULT","error":"nonfinite",
    "current":false}
17 {"type":"sample","synthetic":true,"uptime_ms":6000,"temperature_c":25.00,"state":"FAULT","error":"recovering",
    "current":true}
18 {"type":"sample","synthetic":true,"uptime_ms":7000,"temperature_c":25.00,"state":"NORMAL","error":"none","current":true}
19
20 ```
21 ### stderr
22 ```text
23
24
25
26 ## M06-L03-changed-stale
27 Actual exit: 0
28
29 ### stdout

```

SS13-04. Actual changed normal replay: eight synthetic sample records, including recovery and the nonfinite fault.


```

1  # Actual CP05 changed execution results
5  ## M06-L03-changed-normal-v02
8  ### stdout

19
20 ```
21 ### stderr
22 ```text
23
24 ```
25
26 ## M06-L03-changed-stale
27 Actual exit: 0
28
29 ### stdout
30 ```text
31 {"type":"startup","version":"0.5.0","synthetic":true,"input_status":"missing"}
32 [{"type":"sample","synthetic":true,"uptime_ms":0,"temperature_c":22.00,"state":"FAULT","error":"recovering",
   "current":true}
33 {"type":"sample","synthetic":true,"uptime_ms":1000,"temperature_c":22.00,"state":"NORMAL","error":"none","current":true}
34 {"type":"sample","synthetic":true,"uptime_ms":4000,"temperature_c":22.00,"state":"NORMAL","error":"none","current":true}
35 {"type":"sample","synthetic":true,"uptime_ms":4001,"temperature_c":null,"state":"FAULT","error":"stale","current":false}
36 {"type":"sample","synthetic":true,"uptime_ms":5000,"temperature_c":22.00,"state":"FAULT","error":"recovering",
   "current":true}
37 {"type":"sample","synthetic":true,"uptime_ms":6000,"temperature_c":null,"state":"FAULT","error":"missing",
   "current":false}
38 {"type":"sample","synthetic":true,"uptime_ms":7000,"temperature_c":22.00,"state":"FAULT","error":"recovering",
   "current":true}
39 {"type":"sample","synthetic":true,"uptime_ms":8000,"temperature_c":22.00,"state":"NORMAL","error":"none","current":true}
40
41 ```
42 ### stderr
43 ```text

```

SS13-05. Actual stale replay: expiry and missing input produce FAULT with current false, followed by recovery.

LAB13 - Expose freshness without changing policy

Allow forty-five to sixty minutes. Record your clean CP05 baseline, create a branch if you initialized a local practice repository, and apply the bounded reporting change. Keep the original checkpoint ZIP available for recovery. If you do not yet have a Git repository in this exercise folder, compare the changed files with a separately extracted baseline rather than pretending an absent history exists.

Run normal and stale-recovery fixtures after the change. Inspect the full JSON lines first. Then use PowerShell's JSON parser to inspect the new field without relying on visual alignment:

```
$lines = .\scripts\replay.cmd fixtures/stale-recovery.csv
$runExit = $LASTEXITCODE
$records = $lines | ForEach-Object { $_ | ConvertFrom-Json }
$samples = $records | Where-Object { $_.type -eq 'sample' }
$samples | Format-Table uptime_ms, state, error, current
```

A parsing error matters; do not discard it because some lines look correct. For the stale fixture, expected current flags are true, true, true, false, true, false, true, true. In particular, the 5000 and 7000 recovery records have current data while still in FAULT. The new field must express the snapshot meaning, not simply test whether the state equals NORMAL.

Cross-compile the modified shared source with the unchanged manifest target. Save its actual exit and output separately from the host result. If the compiler or package is unavailable, record that exact open check. Do not substitute an older firmware file or a reassuring agent response.

Review the diff last as well as first. The intended implementation changes the status formatter and its documentation. A changed recovery counter, delay, FQBN, or library dependency requires an explanation against the brief and usually indicates scope drift. Remove unrelated edits before saving a teaching commit.

Failure, diagnosis, and recovery

Suppose an assistant implements current as state != FAULT. The normal case may look fine, but recovery records become false even though they contain fresh numeric readings. The stale-recovery fixture exposes that mistake without a board. Correct the mapping to the existing snapshot flag, rerun the same sequence, and preserve the before/after evidence.

Suppose the output contains "current": "true". That is a string, not a Boolean. A human can read it, but a machine consumer may treat it differently. Parse the JSON and inspect the field's type rather than merely searching for the word true. Stable machine-readable formats are contracts between programs, not just attractive terminal text.

If a patch causes widespread compiler errors, inspect the small format-string edit before adding dependencies or replacing the project. Restore the formatter from the known baseline if necessary, confirm the baseline still runs, and reapply the change in a smaller step. A recoverable failure is part of the workflow; a hidden failure is an evidence problem.

Worked answer and questions

A successful solution reads `Snapshot.current` directly, emits JSON Boolean literals, retains all existing sample fields, and leaves policy and acquisition unchanged. Its stale-recovery output matches the eight expected flags and preserves the previous state/error sequence. Its report identifies the source change, host results, firmware result or open gate, and unperformed physical checks.

Why can FAULT coexist with a current temperature? Recovery requires two consecutive valid samples. The first is fresh data, but the policy deliberately has not returned to NORMAL. Freshness and readiness are related but distinct properties.

What does a firmware build prove here? It establishes that the selected toolchain compiled the current source for the recorded target. It does not establish upload, USB enumeration, actual sensor input, or physical timing.

Why does the `nan` fixture scenario exit zero? The file is valid and the expected policy behavior is a nonfinite fault. A process failure would be a different event, such as a malformed row or failed compilation.

Why use the shared source on the host? It allows controlled input to exercise the actual policy being compiled into firmware. A separately reimplemented “simulation” could pass while the production policy retained the defect.

Transfer and evidence to keep

Read a saved log with the next maintainer in mind. A filename such as `output.txt` says little about its input or source. A short accompanying note can identify CP05, the exact fixture, the compiler, the formatter change, and the run's exit without putting personal machine details into the sample output. Keep configuration and evidence identity outside the stable sample schema unless consumers actually need them in each record.

Also preserve the distinction between an expected transcript and captured output. The state sequence printed in this chapter explains the required behavior; your terminal run establishes what your local executable did. If a result differs, retain the difference and investigate it. Replacing the captured text with a reference sequence would erase the information you need to debug. A useful handoff can include both, labeled clearly, with a short explanation of any unresolved mismatch.

Choose a reporting change in your own project that can be made without changing acquisition. Define its field name, type, meaning, and expected boundary behavior. Identify a sequence that would expose a misleading implementation, then ask for the smallest patch and inspect its actual output.

Keep the brief, reviewed diff, fixture identity, tool versions, command exits, and a short limitation statement together. This evidence is small enough to understand and complete enough to reproduce. Chapter 14 will use the same discipline when the starting behavior is wrong and several explanations compete.

14 - Debug with Evidence

Debugging starts with a difference between what should happen and what actually happened. A useful assistant helps make that difference smaller and more precise: it reads the relevant source, proposes explanations, and selects a test whose outcome distinguishes them. It is less useful when it responds to an incomplete symptom with a large patch and a confident story.

This chapter uses CP06 and the separate fault checkpoints. You will diagnose a malformed synthetic fixture, then investigate a stale-reading defect. Both can be completed without hardware. The aim is to show the evidence that supports a cause, apply a bounded correction, and demonstrate the original reproducer working afterward. Chapter 15 adds a fuller native test workflow.

Write the symptom before naming the cause

“The sensor is broken” describes neither the observed behavior nor the environment. “The replay exits two at fixture line three when the value is 28C” gives another person something to reproduce. It also avoids implying that a physical sensor was involved when the input came from a file.

Create a short record with five items: starting checkpoint, exact command, expected behavior, actual output/exit, and the smallest input that produces the difference. Add a source hash or commit identity when available. If you changed several files before the failure, include the diff instead of reconstructing the change from memory.

Separate expectation from observation. You might expect a valid numeric sample at timestamp 1000, while the observed adapter error says the supplied token is invalid. Record both. Do not silently edit the fixture and then describe the corrected file as the original reproducer. The history of the experiment matters because it explains what the correction changed.

An assistant can help format this record, but it cannot supply missing output that was never captured. If a terminal session has disappeared, rerun the command in a disposable copy. A recreated run is useful evidence when labeled honestly. It is not the original live chronology.

Locate the failure layer

Before interpreting policy, ask whether the source compiled, the executable started, and the input was accepted. A compiler error means source could not be translated successfully. A linker error means compiled pieces could not be combined as required. A process that cannot start may have a runtime dependency problem. A running process can then reject input or implement the wrong policy.

The layers suggest different next checks. If the compiler reports a missing header, inspect the include path and source layout. If a native executable exits before printing anything, capture its exact exit and inspect the documented runtime DLL path. If JSON output appears and then a line-number error occurs, the process has already started and reached input handling.

A real Windows rehearsal of this project exposed that distinction. The same native test executable initially exited without stdout when its runtime DLL directory was absent from the process PATH. With the compiler distribution's bin directory available to that process, the same executable passed its assertions. The evidence supported an execution-environment cause, not a policy rewrite. The supplied .cmd wrappers account for this process-local dependency.

Do not turn that example into a universal diagnosis for every silent program. It illustrates how to preserve the executable, change one relevant condition, and compare results. A different failure needs its own evidence. Similarly, a remote-control tool failing to start a guest session is not automatically a defect in the program it was supposed to run.

Worked example: a malformed input token

Open CP06 in a new folder and inspect `fixtures/invalid.csv`. It is explicitly labeled as a malformed synthetic fixture. The first data row supplies a valid number. The second supplies 28C, which does not match the adapter's numeric grammar.

Run the reproducer:

```
.\scripts\replay.cmd fixtures/invalid.csv
$invalidExit = $LASTEXITCODE
```

The expected result is process exit two and a diagnostic identifying line three as an invalid sample number. A startup record and the earlier valid sample can already have been printed. Partial output does not mean the rejected token became a valid temperature. Read both stdout and stderr, and keep the line number with the actual file revision.

Now write three candidate explanations before changing anything: the fixture contains an invalid token; the parser rejects an otherwise supported number; or the policy fails after receiving a valid reading. Each explanation points to different evidence. The token in the file, the parser contract, and the point at which the error occurs help distinguish them.

Inspect `host/replay.cpp`. The adapter accepts a numeric value only when the whole token is consumed and the result is appropriate for an ordinary sample. It also has explicit special tokens for modeled missing and nonfinite input. The suffix C is not a unit annotation supported by this fixture format. Temperature units are part of the documented field meaning, not a suffix to append to each token.

Choose the smallest discriminating test

Make a copy of the fixture named `fixtures/lab14-valid.csv`. Change only 28C to 28. Run that copied fixture through the same executable path. If the valid token is accepted and the modeled sample reaches policy, the result supports the input-format explanation. You have changed the suspected cause while holding the toolchain and policy constant.

Do not “fix” the example by teaching the parser to ignore arbitrary suffixes. That changes the accepted language and can turn malformed input into a plausible reading. It also prevents the

experiment from answering whether the original fixture violated the existing contract. A format change could be a legitimate future task, but it would need its own requirements and tests.

Do not replace the token with zero merely to avoid an error. Zero is a valid numeric measurement with a meaning different from missing or malformed input. Inventing a temperature hides the failure. If the scenario intends missing input, use the documented missing event representation; if it intends a numeric sample, supply that number.

After the copied fixture succeeds, rerun the original `invalid.csv` and confirm it still rejects the malformed token. That second check establishes that the correction did not weaken the parser. Preserve both command records: accepted corrected input and rejected original invalid input. Together they support a more precise conclusion than one passing run alone.

Use AI to compare explanations

Give the assistant the exact symptom record, the small fixture, relevant parser code, and the format requirement. Ask it to propose a discriminating test before editing source. For example:

The CP06 replay rejects line three of this synthetic fixture with exit two. The row contains 28C. Explain two plausible failure locations using the supplied adapter and policy code, then choose one small test that distinguishes them. Do not change the parser contract or invent a temperature. After the test, state what its result supports and what remains untested.

The assistant's proposal should describe a predicted observation for each explanation. “Try changing the input” is incomplete unless it says which change, which command, and how the result changes the diagnosis. The value of the request is that it makes the reasoning operational and reviewable.

When the result arrives, update the diagnosis. If the corrected token still fails, do not repeat the same edit in a different file. Check whether the command used the copied fixture, whether you are in the intended project folder, and whether the executable was rebuilt from the intended source. Evidence can contradict a plausible first explanation; that is progress when you preserve it.

A second defect: stale data treated as current

Now use the separate `FAULT-stale` checkpoint in another exercise folder. It is based on the CP07 normal/fault implementation and includes tests that Chapter 15 will explain. For this chapter, use its replay path and source inspection; you do not need to understand the full test harness yet.

The supplied mutation disables the expiry condition in `Monitor::poll`. A reading that should become stale can remain current. This is a different kind of failure from malformed input: the file is valid, the executable runs, and the output is valid JSON. The meaning of the output is wrong.

Run `fixtures/stale-recovery.csv`. At timestamp 1000, the last valid sample is current. At 4000 its age is exactly 3000 milliseconds, so it should still be current. At 4001 the age is 3001, and the

correct output is FAULT/stale with a null temperature. The defective version fails that expected transition.

Write competing explanations: the timestamps were parsed incorrectly; the expiry condition did not run; or the formatter displayed an old value despite a correct policy state. Inspect the output's timestamp, state, error, and temperature together. A number alone cannot tell you which layer failed.

Trace the stale path without broad changes

Locate the call to `monitor.poll(now)` in the host adapter, then inspect the policy's `poll` method and the fail helper. The helper is significant: it sets FAULT, stores the error, marks the value unavailable, and resets the recovery count. A correct expiry path needs the whole transition, not just a different word in the output.

The fault checkpoint inserts a false condition into the expiry guard. In a review, name the precise trigger and consequence: when a current reading becomes older than the freshness limit, the disabled guard prevents `fail(Error::Stale)` from running. That explanation is supported by the code and the reproducer, rather than by a general suspicion about timing.

A tempting patch changes the printed state to FAULT whenever the timestamp looks old. That can hide the visible symptom while leaving `current_` and the recovery counter wrong. The program might continue emitting a numeric stale value or recover incorrectly. Keep state transitions in the policy and representation in the formatter.

Restore the intended expiry guard in the disposable fault copy. Do not change the threshold constant, fixture timestamps, output schema, or recovery requirement. Then run the original stale-recovery sequence again. Check both the failing 4001 event and the later recovery events, because a patch can correct expiry while damaging what happens afterward.

```

1  # Actual original fixture diagnostic
2
3  Command: .\scripts\replay.cmd fixtures/invalid.csv
4  Actual process exit: 2
5
6  ## Original fixture - unchanged
7  ```csv
8  # INTENTIONALLY MALFORMED SYNTHETIC FIXTURE. Expected process exit 2.
9  0,sample,22
10 1000,sample,28C
11
12 ```
13
14 ## Actual stdout
15 ```text
16 {"type":"startup","version":"0.5.0","synthetic":true,"input_status":"missing"}
17 {"type":"sample","synthetic":true,"uptime_ms":0,"temperature_c":22.00,"state":"FAULT","error":"recovering"}
18
19 ```
20
21 ## Actual stderr
22 ```text
23 replay.cmd : ERROR: fixture line 3: invalid sample number
24 At line:24 char:1
25 + & '.\scripts\replay.cmd' fixtures/invalid.csv 1> $o 2> $r
26 + ~~~~~
27 + CategoryInfo          : NotSpecified: (ERROR: fixture ...d sample number:String) [], RemoteExce
28 + ption
29 + FullyQualifiedErrorId : NativeCommandError
30
31
32 ```

```

SS14-01. Actual original CP06 fixture: the 28C token on line 3 produces invalid sample number and process exit 2.

```

1  # Actual read-only AI hypothesis response
2
3  The original task made no edits and ran no tests. The predictions below are not executed results.
4
5  The evidence points to the intentionally malformed `28C` token in [fixtures/invalid.csv](C:/Users/user/Documents/
  AI-Embedded-Book/capture-practice-m06-v01/m06-l04/sensor-monitor/fixtures/invalid.csv). No files were changed and no
  reproducer or test was run.
6
7  Before any edit, I would consider these candidate failure locations:
8
9  1. **Strict numeric validation in `temperature()`**
10
11  `std::strtof` parses the numeric prefix `28`, but leaves `C` unconsumed. [host/replay.cpp](C:/Users/user/Documents/
  AI-Embedded-Book/capture-practice-m06-v01/m06-l04/sensor-monitor/host/replay.cpp) requires the parse endpoint to
  equal the end of the field, so it throws `invalid sample number`.
12
13  Discriminating test:
14
15  - Copy `invalid.csv` to a temporary fixture, preserving the original.
16  - Change only line 3 from `1000,sample,28C` to `1000,sample,28`.
17  - Command: `.\scripts\replay.cmd <temporary-fixture>`
18  - Predicted observation: exit `0`; startup plus two sample records; the second sample reports `temperature_c:28.00`
    and reaches `NORMAL` after the two valid recovery readings.
19
20  2. **A trailing carriage return reaching the numeric parser**
21
22  With CRLF input, an unremoved `\r` would make an otherwise valid `28` fail the same full-token check. The replay
  loop explicitly removes one trailing `\r`, so this candidate should be rejected by a line-ending comparison.
23
24  Discriminating test:
25
26  - Create two temporary copies containing `1000,sample,28`, one encoded with LF and one with CRLF. Keep the original
  
```

SS14-02. Actual read-only AI response proposes a copy-only numeric-token test and explicitly labels the result as predicted.


```

C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m06-v01\m06-l04\sensor-monitor
.\scripts\replay.cmd fixtures/lab14-valid.csv
{"type":"startup","version":"0.5.0","synthetic":true,"input_status":"missing"}
{"type":"sample","synthetic":true,"uptime_ms":0,"temperature_c":22.00,"state":"FAULT","error":"recovering"}
{"type":"sample","synthetic":true,"uptime_ms":1000,"temperature_c":28.00,"state":"NORMAL","error":"none"}
actual_exit=0
=== M06-L04-original-again.ps1 ===
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m06-v01\m06-l04\sensor-monitor
.\scripts\replay.cmd fixtures/invalid.csv
{"type":"startup","version":"0.5.0","synthetic":true,"input_status":"missing"}
{"type":"sample","synthetic":true,"uptime_ms":0,"temperature_c":22.00,"state":"FAULT","error":"recovering"}
replay.cmd : ERROR: fixture line 3: invalid sample number
At line:24 char:1
+ & '.\scripts\replay.cmd' fixtures/invalid.csv 1> $o 2> $r
+ ~~~~~
+ CategoryInfo          : NotSpecified: (ERROR: fixture ...d sample number:String) [], RemoteException
+ FullyQualifiedErrorId : NativeCommandError

actual_exit=2
=== M06-L04-fixture-diff.ps1 ===
diff --git "a/C:\Users\user\Documents\AI-Embedded-Book\capture-evidence-m06-v01\m06-l04\M06-L04-numeric-copy-before.bin"
      "b/C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m06-v01\m06-l04\sensor-monitor\fixtures\lab14-valid.csv"
index b92bb60..21f89a5 100644
--- "a/C:\Users\user\Documents\AI-Embedded-Book\capture-evidence-m06-v01\m06-l04\M06-L04-numeric-copy-before.bin"
+++ "b/C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m06-v01\m06-l04\sensor-monitor\fixtures\lab14-valid.csv"
@@ -1,3 +1,3 @@
 # INTENTIONALLY MALFORMED SYNTHETIC FIXTURE. Expected process exit 2.
 0,sample,22
-1000,sample,28C
+1000,sample,28
diff_exit=1 (1 means files differ)
ACTUAL_PAIRED_BATCH_COMPLETE

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m06-v01\m06-l04\sensor-monitor>

```

SS14-03. The corrected copy exits 0 and reaches NORMAL; the unchanged original still exits 2. The diff shows the single numeric-token change.

The screenshot shows a VS Code workspace named 'm06-l04 (Workspace)'. The Explorer sidebar on the left displays a file tree with folders like 'sensor-monitor', 'capture-views', 'docs', 'firmware', 'fixtures', 'host', 'scripts', and 'tools'. The 'investigation.md' file is selected and open in the main editor. The editor shows a markdown document with the following content:

```

1  # Completed investigation - M06-L04
2
3  Scope: CP06 synthetic fixture grammar. This is an instructor investigation, not a physical sensor repair.
4
5  ## Symptom and paired experiment
6  - Original command: .\scripts\replay.cmd fixtures/invalid.csv
7  - Actual original result: exit 2; fixture line 3: invalid sample number.
8  - Copied invalid.csv to lab14-valid.csv; changed only 1000,sample,28C to 1000,sample,28.
9  - Copy command: .\scripts\replay.cmd fixtures/lab14-valid.csv
10 - Actual copy result: exit 0; two samples, FAULT/recovering then NORMAL at 28.00 C.
11 - Original repeated unchanged: exit 2 with the same line 3 diagnostic.
12 - Actual diff: one numeric token changed in the copy; diff exit 1 means different files.
13
14 ## Supported cause and scope
15 strtouf accepts the numeric prefix 28 but leaves C unconsumed. The existing full-token check rejects that suffix before
16 the rejected sample reaches policy. No parser weakening was needed.
17
18 The actual read-only AI response proposed discriminating tests. Its line-ending and wrapper-selection alternatives were
19 predictions, not additional executed tests. Only the paired experiment above was performed here.
20
21 All original CP06 source files match their checkpoint hashes, including parser, policy and invalid.csv. The only extra
22 fixture is lab14-valid.csv. No firmware build, board upload or physical measurement is claimed for this investigation.
23
24 ## Evidence handoff
25 Preserved M06-L04-invalid, M06-L04-copy and M06-L04-original-again stdout/stderr/exit files; actual-paired-batch01.txt;
26 original/copy bytes; and the read-only AI task transcript.
27

```

The status bar at the bottom indicates 'Ln 1, Col 1', 'Spaces: 4', 'UTF-8', 'LF', and 'Markdown'.

SS14-04. Completed instructor investigation records the supported cause, paired evidence, preserved sources and limits of the synthetic test.

LAB14 - Produce a defensible fault report

Allow about one hour. Submit two short records: the malformed-fixture diagnosis and the stale-policy diagnosis. Each record needs the starting state, original reproducer, competing explanations, chosen test, observed result, bounded correction, regression check, and remaining limits.

For the fixture record, retain `invalid.csv` unchanged and create the corrected copy. Show that the copied numeric row succeeds while the original malformed row still exits two. Explain why you did not change the parser or substitute a made-up temperature. Name the exact accepted token and its documented unit meaning.

For the stale record, begin from `FAULT-stale`, not a previously repaired folder. Capture the bad 4001 observation before editing. Correct the disabled expiry guard, preserve the diff, and rerun the same fixture. Expected corrected states are `FAULT`, `NORMAL`, `NORMAL`, `FAULT`, `FAULT`, `FAULT`, `FAULT`, `NORMAL`.

The 6000 missing event must interrupt recovery. At 7000, one valid reading is insufficient; only the next valid reading at 8000 allows `NORMAL`. Include those observations in the regression record. Checking only the first stale transition would leave an important part of the correction unexplored.

Cross-compile the repaired shared source with the unchanged target if the recorded toolchain is available, and save the actual result. A host correction and a firmware compile answer separate questions. If the current build has not been performed, leave that box open and identify the command still required.

Ask an assistant to draft the report from your raw evidence, then review its verbs. Replace “proved the sensor works” with a claim supported by the run, such as “the shared policy expired a synthetic sample at age 3001 milliseconds.” Remove a root-cause statement if the chosen test never actually distinguished the competing explanations.

Worked answer and recovery route

The fixture failure is caused by a token outside the defined input format. Correcting only the token in a copied fixture resolves that scenario while preserving strict rejection of the original. The stale failure is caused by a disabled expiry guard; restoring the policy transition corrects availability and recovery state. These conclusions are narrow enough to reproduce.

A good report includes failure evidence as well as the fixed result. A screenshot of a passing terminal alone cannot establish what was wrong or whether the original case was exercised. Keep the small text log and source diff even when you also capture a full screenshot for explanation.

If you lose track of edits, preserve the current folder and extract the fault checkpoint into a new one. Compare the files and repeat the original command. Do not delete useful work merely to get a clean-looking result. When Git is present, use the `restore` or `revert` operation appropriate to an uncommitted or committed change, as practiced in Chapter 7.

Review questions

Why record the process exit separately from the state field? FAULT can be the correct application output from a successful replay. A nonzero process exit describes an execution or input problem in a different channel.

What makes a test discriminating? Different plausible explanations predict different observations under a controlled change. A test that produces the same result under every explanation does little to select among them.

Why can changing only the output label be a bad stale-data fix? It can leave the policy's availability and recovery state incorrect. The visible word improves while the underlying transition remains broken.

When may you call an explanation a root cause? When the relevant evidence supports the mechanism, a bounded correction addresses it, and the original reproducer plus meaningful regression checks behave as required. A plausible narrative alone is insufficient.

Would an oscilloscope help with this fixture typo? It would not test the text parser's accepted language. Instruments are valuable when the uncertainty concerns physical signals or timing; choose evidence appropriate to the question.

Transfer to a real maintenance task

Decide when the investigation is finished before adding more experiments. For these exercises, completion means the original case is explained by a supported mechanism, the smallest correction addresses that mechanism, and the relevant nearby behavior still works. It does not require proving every possible property of the program. An assistant that keeps proposing unrelated improvements after those checks can make the result harder to review.

Write down useful questions that fall outside the bounded fix without silently treating them as new requirements. A physical driver's timeout, for example, may deserve a later task, but it does not explain why a CSV token contains a letter. This separation keeps the report accurate about both what was resolved and what remains. It also lets another engineer resume an unresolved question without repeating a completed diagnosis.

Take a recent failure from your own project and rewrite its description without a proposed cause. Include exact input, output, environment, and expectation. Name two explanations and one observation that would distinguish them. If you cannot reproduce the failure, record what information is missing before changing production code.

Use the assistant to reduce the experiment, not to conceal uncertainty. A small failed test can be more useful than a large speculative patch because it tells you which path deserves attention. The next chapter makes these checks repeatable and demonstrates that a test suite can reject a deliberately wrong change.

15 - Test, Review, and Measure

A passing test is useful when you can explain what would make it fail. This chapter turns the normal/fault behavior from the previous chapters into repeatable checks and then deliberately breaks one comparison. You will see a successful compile followed by meaningful assertion failures, restore the source, and run the same tests again. The exercise shows what the tests establish and where their evidence ends.

Use CP07 in a fresh exercise folder. This checkpoint contains 36 native assertions for the already introduced normal/fault policy, freshness, recovery, scheduling, rollover, and basic output formatting. It does not contain the configurable threshold, hysteresis, or command parser from the Chapter 17 capstone. Do not import those later acceptance tests into an earlier lesson and treat their absence as a defect.

Begin with expected behavior

Before opening the test implementation, read `docs/requirements.md`. Write three predictions in plain language: a sample exactly 3000 milliseconds old remains current; a sample 3001 milliseconds old becomes stale; and one fresh valid sample after a fault is insufficient for recovery. These statements give the test an independent source of expected behavior.

Now predict an entire short sequence. Start FAULT/missing. Supply valid readings at zero and 1000: the first begins recovery and the second allows NORMAL. Poll at 4000: the last reading is exactly 3000 milliseconds old and remains current. Poll at 4001: it becomes FAULT/stale with no current temperature. The expectation comes from the requirement, not from copying the comparison already in the source.

This independence is easy to lose when asking an assistant for tests. If the prompt only says “test this code,” the answer may encode the current implementation's assumptions. Give the requirement and the code, ask for a case that would distinguish nearby wrong implementations, and review the expected value separately from the mechanics of the test.

An independent expected value need not come from a second person. It can come from a requirement table, an explicit state transition, or a separately reasoned example. The point is to avoid using the same expression to calculate both the implementation's result and the test's expected result.

Execute the actual shared policy

Open `scripts/common.cmd`. The native build lists the policy and output source files from `firmware/sensor-monitor`. Open the Arduino sketch directory and confirm those are the same files compiled into the sketch. The host test does not maintain its own alternative version of normal/fault behavior.

Run the test wrapper from PowerShell:

```
.\scripts\test.cmd
$testExit = $LASTEXITCODE
```

For the unmodified CP07 checkpoint, the recorded host reference result is PASS 36 assertions; 0 failures; synthetic host tests, with exit zero. Your actual run should be saved with its compiler version and source identity. If a different checkpoint reports a different count, inspect the stage before deciding whether something failed. CP09's full capstone suite has 114 assertions and covers additional behavior.

The wrapper compiles before executing. A compiler failure should stop the process before a test result is claimed. Likewise, an executable that cannot start is not a passing test with missing text. Record the exit and diagnose the documented compiler/runtime location. The .cmd wrappers use a process-local PATH for the native runtime and require no PowerShell execution-policy change.

Read the output, not just the color of a terminal badge. A nonzero exit and a named failing assertion tell you which observation disagreed with the expectation. A compiler error, loader error, failed assertion, and intentionally rejected fixture are distinct events. Keeping their outputs together but clearly labeled makes later diagnosis much easier.

Read a small test as a story

In tests/test_monitor.cpp, find the basic test function. It creates a fresh Monitor, checks its starting state, supplies readings, advances the injected clock, and inspects snapshots. The sequence is short enough to read beside the requirement. No physical sensor or serial connection is needed to produce its inputs.

The CHECK macro increments an assertion count and records a failure when an expression is false. Its line-number report is a pointer into the exact test source used for that run. Line numbers may change after you add a test, which is why the source identity belongs in the evidence record.

Consider these calls from the sequence:

```
p.sample(1000, 21.0f);
p.poll(4000);
CHECK(p.snapshot(4000).current);
p.poll(4001);
CHECK(p.snapshot(4001).error == Error::Stale);
```

The numbers are injected timestamps. The test need not wait three real seconds between calls. It asks whether the policy computes the required age and transition for those inputs. It does not measure how promptly an MCU interrupt or USB transaction would occur.

Notice that the test checks both availability and error meaning. A weaker test might check only that a line was printed. That would miss a stale numeric value mislabeled as usable. Choose observations that distinguish the required behavior from a plausible wrong implementation.

Cover failure and recovery together

Invalid-input tests should check what happens after the fault as well as when it begins. CP07 supplies missing input, NaN, and positive infinity. It confirms that invalid data is not current, then exercises valid/invalid recovery sequences. A later valid sample cannot simply erase all preceding state.

Recovery begins with a fresh valid reading but remains in FAULT. Another invalid reading resets the count. Two consecutive fresh valid readings then allow NORMAL. A long stale gap between otherwise valid readings also breaks the sequence. These cases guard against a counter that continues accumulating “good” samples across unrelated failures.

The test should make that distinction visible in its inputs. “Recover after a fault” is too broad if it only sends two valid samples immediately after one missing event. Include the interruption that an incorrect implementation could mishandle. Each additional case earns its place by distinguishing a meaningful error.

Do not turn every possible float value into a separate named test. Select classes of behavior: finite values, missing data, nonfinite results, freshness equality, stale age, and recovery interruption. Within a class, choose examples that exercise an important boundary. This keeps the suite understandable while protecting the decisions the project actually makes.

Understand rollover arithmetic

The clock values are unsigned 32-bit integers. The implementation calculates elapsed time by unsigned subtraction. Under the documented chronological-call assumptions, that handles a single wrap from a large timestamp back to zero. The assumption about call ordering and maximum gap remains part of the contract; arithmetic alone cannot infer an arbitrary number of missed wraps.

CP07 tests a reading at a timestamp one second before wrap and another at zero. It then checks current data at 3000 and stale data at 3001. These cases prove the implemented arithmetic for the injected sequence. They are much more useful than an unsupported comment saying “rollover safe.”

The scheduler has related checks: immediately due at startup, not due too early, due at the interval boundary, and only one acquisition for a delayed turn. A long pause does not cause several identical catch-up acquisitions at one timestamp. That behavior is a design choice which the tests make explicit.

If you transfer this pattern to a different clock type, revisit its width, signedness, ordering assumptions, and expected maximum gap. Do not paste an unsigned-subtraction idiom into code whose inputs have a different meaning. Ask the assistant to explain those assumptions before proposing a portability change.

Prove a boundary test can reject a defect

Make a new disposable CP07 copy for the mutation exercise. In `Monitor::poll`, the correct guard expires a sample when its age is greater than `kStaleAfterMs`. Change only that comparison from `>` to `>=`. Do not change the requirement or its expected test result to agree with the defect.

This mutation is deliberately small and compiles successfully. It makes the policy expire a reading at exactly 3000 milliseconds, one millisecond earlier than specified. A test that checks only 2999 and 3001 would miss it. The equality case exists to distinguish these nearby implementations.

Run the same test command. In the verified instructor host exercise, the mutation compiled, then the 36-assertion suite exited one with two failures: the ordinary equality check and its rollover counterpart. Restore the original comparison, rebuild, and run again; the same suite returns to zero failures. Those before/after receipts are software evidence from injected inputs, not a physical timing measurement.

The number of failures is secondary to their meaning. Another test layout may report the same defect with a different count. What matters is that the unmodified requirement-based test rejects the wrong behavior, that the failure is not merely a syntax error, and that restoring the source resolves the original case.

Review memory and output boundaries

The policy uses a small fixed state: the current temperature, timestamp, state/error values, recovery count, and availability flag. Inspect how failure changes those values together. A stale transition that changes only the state enum can leave a numeric value marked current, which is why the tests inspect more than one field.

The formatter writes into a bounded buffer and checks whether the complete output fits. CP07 includes a deliberately small buffer case that must report failure. This is a functional check on the formatting API's contract, not a measurement of total firmware memory use. The native executable's address space does not resemble the MCU's complete memory layout.

When reviewing a buffer change, examine both capacity and result handling. A larger buffer can hide a truncation symptom while callers continue ignoring a failed formatting result. A smaller buffer can be correct if the maximum output and failure behavior are understood. The source and tests should explain why the chosen size is sufficient for the current schema.

The capstone later introduces a bounded command buffer. Do not add a parser to CP07 merely to create more memory tests. Test the boundaries of the functionality present at this stage, then extend the suite with the feature that creates a new risk.

Combine tests, replay, and compilation

Native assertions are precise but terse. Replay output provides a readable sequence connecting timestamps, temperatures, states, and errors. Use both: run the basic suite, then run `fixtures/stale-recovery.csv` and explain the 4000, 4001, 6000, 7000, and 8000 events in ordinary language.

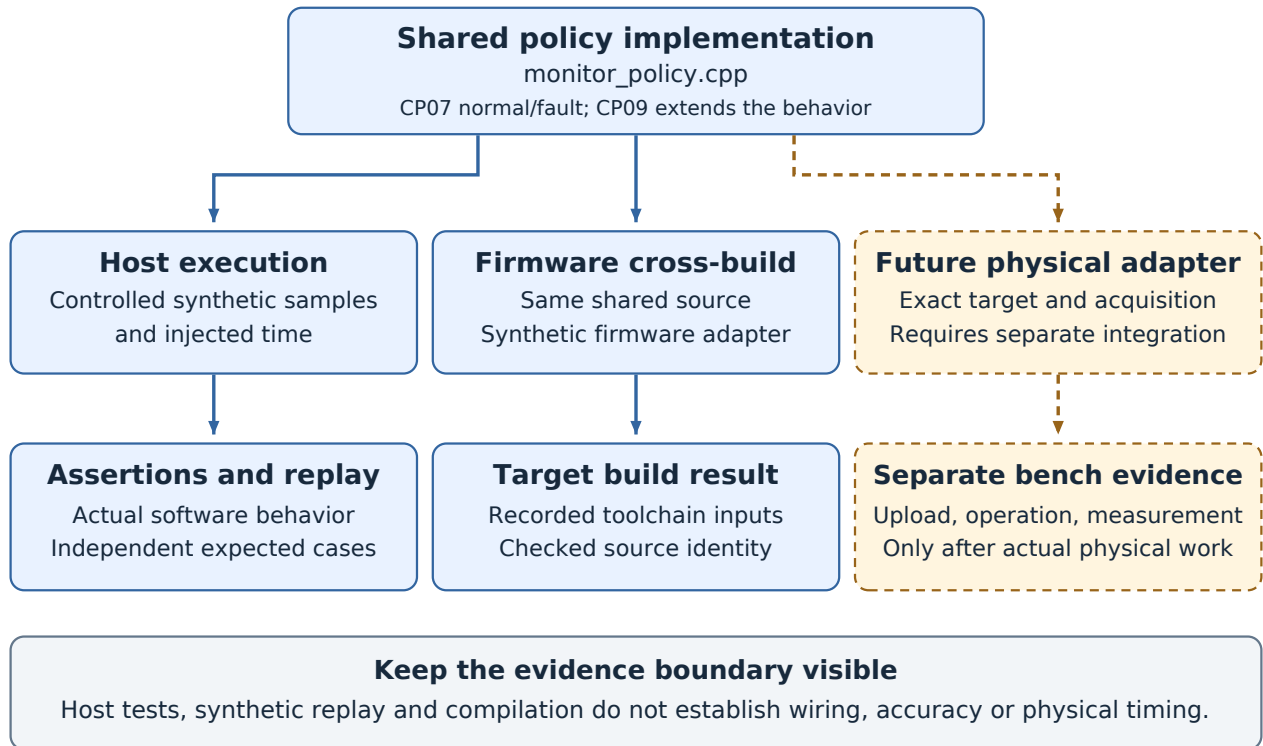
Cross-compilation asks another question: does this source build for the selected firmware target with the recorded dependencies? Save the actual command, full target, exit, and artifact identity. A native pass cannot substitute for that build, and a firmware build cannot substitute for the native behavior checks. The final CP09 Windows fresh-clone build exits zero, with 362,508 program bytes and 54,952 global-variable bytes; those measurements identify that release and do not substitute for this CP07 mutation exercise's build.

Physical timing, sensor accuracy, electrical compatibility, and upload/serial behavior need evidence of their own. List those as unperformed when no bench session occurred. You can complete the core course without hardware, but you cannot relabel a software run as a physical observation to make the report look complete.

An optional automation can run the same commands after a source change. Keep it thin: checkout, identify dependencies, compile, execute, preserve results, and fail on a failed required command. Automation is valuable because it repeats a known check, not because a green badge makes a weak check stronger. Do not add accounts, paid runners, or a new framework merely for this lesson.

One policy source, distinct verification paths

D12 · Conceptual diagram · Physical extension is optional



D12. Shared policy supports host checks and synthetic firmware compilation. A physical adapter needs separate integration and bench evidence.

```

6  #include <limits>
7  #include <string>
8
9  using namespace sensor;
10 int failures = 0;
11 int assertions = 0;
12 #define CHECK(c) do { ++assertions; if (!(c)) { ++failures; std::cerr << "FAIL line " << __LINE__ << ": " #c "\n"; } } whi
13
14 void basic() {
15     Monitor p;
16     CHECK(p.snapshot(0).state == State::Fault);
17     CHECK(p.snapshot(0).error == Error::Missing);
18     CHECK(!p.snapshot(0).current);
19     p.sample(0, 20.0f);
20     CHECK(p.snapshot(0).error == Error::Recovering);
21     p.sample(1000, 21.0f);
22     CHECK(p.snapshot(1000).state == State::Normal);
23     p.poll(4000); // Exactly 3000 ms remains current.
24     CHECK(p.snapshot(4000).current);
25     p.poll(4001);
26     CHECK(p.snapshot(4001).state == State::Fault);
27     CHECK(p.snapshot(4001).error == Error::Stale);
28     CHECK(!p.snapshot(4001).current);
29     p.sample(5000, 20.0f);
30     CHECK(p.snapshot(5000).state == State::Fault);
31     p.sample(6000, 20.0f, false);
32     CHECK(p.snapshot(6000).error == Error::Missing);
33     p.sample(7000, 20.0f);
34     CHECK(p.snapshot(7000).state == State::Fault);
35     p.sample(8000, 20.0f);
36     CHECK(p.snapshot(8000).state == State::Normal);
37     p.sample(9000, std::numeric_limits<float>::quiet_NaN());

```

SS15-01. Independent CP07 assertions test startup, recovery and inclusive stale equality.

```

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-l01\sensor-monitor> $ErrorActionPreference='Stop';
$k='C:\Users\user\Documents\AI-Embedded-Book\capture-plans-m07-m08-v01\commands';$e='C:\Users\user\Documents\AI-Embedded-Book\c
apture-evidence-m07-m08-v01\m07-l01';$out=Join-Path $e 'actual-baseline-batch01.txt';if(Test-Path $out){throw 'Preserve output'
}};$rows=@(@( 'M07-L01-tests.ps1', 'f4d5843720dcf267d3f7b1b9c44e56051cc77157e82c1cc1b58954c7248d4918'),@( 'M07-L01-stale.ps1', '80fb
7d487323f5b5fee1a3bf32c0c137d013c098d53d0ba2843b54c86f908e66'),@( 'M07-L01-write-map.ps1', '9eb8cf1259a05d675233e609e5af33e9856db
c0c3ba68423498638e66805f67d'));&{foreach($row in $rows){if($row[0] -eq 'M07-L01-write-map.ps1'){ $a=(Get-Content (Join-Path $e
'M07-L01-stale\stdout.txt'))|ForEach-Object {$_|ConvertFrom-Json}|Where-Object {$_.type -eq 'sample'}};if($a.Count -ne 8 -or ($a
.state -join ',') -ne 'FAULT,NORMAL,NORMAL,FAULT,FAULT,FAULT,FAULT,NORMAL'){throw 'Unexpected stale sequence'}};$q=Join-Path $k
$row[0];if((Get-FileHash $q -Algorithm SHA256).Hash -ine $row[1]){throw 'Command mismatch'};Write-Output ('== '+$row[0]+' ==
');&([scriptblock]::Create([IO.File]::ReadAllText($q)));Write-Output 'ACTUAL_M701_BATCH_COMPLETE'} *>&1|Out-String -Stream -Wi
dth 160|Tee-Object -FilePath $out
=== M07-L01-tests.ps1 ===

Path
----
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-l01\sensor-monitor
PASS 36 assertions; 0 failures; synthetic host tests
actual_exit=0
=== M07-L01-stale.ps1 ===
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-l01\sensor-monitor

```

SS15-02. Actual guest CP07 host tests pass all 36 assertions with zero failures and exit 0.

```

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-102\sensor-monitor> $ErrorActionPreference='Stop';
$k='C:\Users\user\Documents\AI-Embedded-Book\capture-plans-m07-m08-v01\commands';$out='C:\Users\user\Documents\AI-Embedded-Book
\capture-evidence-m07-m08-v01\m07-102\actual-mutated-batch01.txt';if(Test-Path $out){throw 'Preserve output'};$rows=@(,@('M07-L
02-mutated.ps1','612119fb2071eadda5bb791223177de64d41670e7fc304421a635c48dad3ef56'));&{foreach($row in $rows){$q=Join-Path $k $
row[0];if((Get-FileHash $q -Algorithm SHA256).Hash -ine $row[1]){throw 'Command mismatch'};Write-Output ('=== '+$row[0]+' ===')
;$p='C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-102\sensor-monitor';&([scriptblock]::Create([IO.
File]::ReadAllText($q)))};Write-Output 'ACTUAL_MUTATED_BATCH_COMPLETE'} *>&1|Out-String -Stream -Width 160|Tee-Object -FilePath
$out
=== M07-L02-mutated.ps1 ===

Path
----
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-102\sensor-monitor
FAIL 36 assertions; 2 failures; synthetic host tests
test.cmd : FAIL line 24: p.snapshot(4000).current
At line:32 char:1
+ & .\scripts\test.cmd 1> (Join-Path $o 'stdout.txt') 2> (Join-Path $o ...
+ ~~~~~
+ CategoryInfo          : NotSpecified: (FAIL line 24: p.snapshot(4000).current:String) [], RemoteException
+ FullyQualifiedErrorId : NativeCommandError

FAIL line 53: r.snapshot(3000).current
actual_exit=1
ACTUAL_MUTATED_BATCH_COMPLETE

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-102\sensor-monitor>

```

SS15-03. The deliberately premature stale boundary compiles and causes two meaningful assertion failures, with exit 1.

```

1  ## Actual CP07 boundary mutation evidence
2
3  Requirement: age3000 remains current; age3001 expires.
4  Injected timestamps are controlled test inputs, not measured device latency.
5
6  ## Actual baseline
7  Command: .\scripts\test.cmd
8  Process exit: 0
9  PASS 36 assertions; 0 failures; synthetic host tests
10
11 ## Actual mutated
12 Command: .\scripts\test.cmd
13 Process exit: 1
14 FAIL 36 assertions; 2 failures; synthetic host tests
15
16 ## Actual restored
17 Command: .\scripts\test.cmd
18 Process exit: 0
19 PASS 36 assertions; 0 failures; synthetic host tests
20
21 ## Actual build
22 Command: .\scripts\build-firmware.cmd
23 Process exit: 0
24 Sketch uses 355248 bytes (12%) of program storage space. Maximum is 2883584 bytes.
25 Global variables use 54872 bytes (16%) of dynamic memory, leaving 272808 bytes for local variables. Maximum is 327680
  bytes.
26
27 Mutation failures: ordinary equality line24 and rollover equality line53.
28 Only poll operator > became >=. Assertions remained unchanged.
29 Restored policy SHA256:
30 43BC64E21CEA5AF086DDEB982BF3E57B1D105889C348E199E5C17CD468278088
31 Software checks and target compilation do not establish physical timing or electrical correctness.

```

SS15-04. Actual boundary results preserve the passing baseline, two-failure mutation, restored pass and successful build. Injected timestamps are explicitly distinguished from measured device latency.

```

1  # Software evidence report
3  ## Actual receipts
32 Actual result: See preserved stdout/stderr
33
34 ## Three supported software claims
35 1. CP07 assertions and synthetic stale replay support the defined startup, recovery and inclusive age3000 boundary for
36   their saved source/input. Injected time is not measured device latency.
37 2. The compiled > to >= mutation is rejected at ordinary and rollover equality. Baseline36/0, mutated36/2, restored36/
38   0; assertions unchanged. Saved tiny diff: m07-102/actual-mutation-diff01.txt.
39 3. Restored shared source cross-build supports compilation for the recorded target/toolchain only; no upload or bench
40   result.
41
42 ## Corrected constructed overclaim
43 Original instructional overclaim: replay passed so the board is verified.
44 Correction: synthetic replay supports checked software behavior; physical behavior remains unestablished.
45
46 ## UNPERFORMED physical checks and future evidence
47 Wiring: inspect exact board revision, pinout and continuity; preserve dated setup photos and meter readings.
48 Timing: instrument acquisition and fault response with a logic analyzer; preserve raw trace, clock basis and test
49   conditions.
50 Accuracy: compare the sensor with a traceable reference at documented conditions; preserve raw paired readings and
51   uncertainty.
52
53 ## Caller and process contracts
54 Call poll or sample_at before snapshot when freshness must be updated; snapshot alone does not expire data.
55 Modeled FAULT may accompany replay exit0; malformed grammar uses exit2; compiled assertion rejection uses exit1.
56 Later edits require comparison to saved source inventories and relevant new checks. Missing receipts remain
57   unestablished.

```

SS15-05. The evidence report separates three supported software claims from a constructed overclaim and three explicitly unperformed physical procedures.

LAB15 - Detect and explain a wrong comparison

Allow sixty minutes. Record a clean CP07 test result and the expected equality rule. Create a disposable mutation copy or branch, change only the expiry comparison to $\geq$, and run the unmodified suite. Save the compiler result separately from the failed assertions. Explain which input distinguishes the wrong comparison from the requirement.

Restore the correct source and run the suite again. Inspect the diff to confirm that the mutation is gone. Run the stale-recovery fixture and verify that 4000 remains current while 4001 is stale. Cross-build the restored source when the recorded target toolchain is available and preserve the actual receipt.

Add one independently motivated test of your own. A suitable example is negative infinity: the requirement rejects nonfinite data, while the existing examples already cover NaN and positive infinity. State the expected state, error, and current-data result before writing the assertion. Do not alter production behavior if the new test confirms an already supported case.

Write a verification note with three parts: checks executed and their source identity; observations and their interpretation; remaining physical and environment-specific limits. Include the deliberate mutation failure. A report containing only the final pass omits the evidence that your chosen test could detect the targeted defect.

Worked answer and review questions

The wrong comparison expires a current sample at the inclusive freshness boundary. The test must keep the 3000-millisecond expectation unchanged and reject that implementation. Restoring $>$ returns the suite to the required behavior. The added nonfinite case should enter FAULT/nonfinite and mark temperature unavailable; its assertion supplements the existing class of invalid-input checks.

Why is a compiler error insufficient for the mutation demonstration? It proves the source cannot build, but it does not show that the behavioral test distinguishes a wrong implementation. The seeded comparison must compile and then fail its meaningful assertion.

What is wrong with calculating the expected state using the same production function? The test can reproduce the implementation's error and agree with it. Derive the expectation from the requirement or a separate model of the intended sequence.

Does an injected 3001-millisecond timestamp measure device latency? No. It supplies a value to test arithmetic and policy. Physical response latency needs a defined measurement setup and an observed event.

Is an assertion count a quality score? No. A small suite that detects the relevant fault can be more useful than many assertions checking superficial details. Understand which incorrect behaviors the cases distinguish.

Carry the method forward

When a new test fails, first read its expectation and setup. The implementation may be wrong, but the test may also have supplied an impossible ordering, selected a different checkpoint, or asserted behavior that was never required. Preserve the failure while checking both sides of the contract. The goal is not to defend the test merely because it was written first; the goal is to make the requirement and executable behavior agree for a reason you can explain.

Review test changes with the same care as production changes. Deleting the equality assertion or changing its expected value can make the mutation appear fixed without correcting policy. A passing report after such an edit needs a clear requirement-based justification. When an assistant changes tests and implementation together, inspect the two diffs separately and verify that the original intended behavior has not moved unnoticed.

Choose one boundary in your own project: a buffer length, timer interval, integer range, or recovery count. Write the equality behavior explicitly, then choose values immediately around it. Ask an assistant for a bounded test and review the expected results before accepting the code. Prove the test rejects one plausible wrong implementation in a disposable copy.

Keep the test's purpose readable. A future maintainer should know which requirement would be lost if the case were deleted or weakened. Chapter 16 will package this style of review and evidence into the second skill and a handoff that works without the old chat history.

16. Build Your Reusable Engineering Toolkit

What you will be able to do

You will turn a practiced review procedure into the second project skill, evaluate it with a meaningful defect and clean work, and resume a bounded task without the old conversation. Start with Chapter 15 and CP08. This checkpoint has shared NORMAL/FAULT policy, synthetic replay, and basic assertions; thresholds, hysteresis, and commands arrive in the capstone. The deliverable is a tested review procedure and a handoff supported by actual evidence. No physical board is required.

Keep the parts that earn their maintenance

A useful toolkit is a small collection of things that reduce repeated effort. You already have a README, sourced hardware context, project instructions, a constraints skill, and maintained command wrappers. Each has a distinct job. The next improvement is not to collect more files indiscriminately, but to identify a recurring action whose result you can evaluate.

Consider the work you repeat during a firmware review. You inspect the requested diff, read the requirement it affects, look at nearby callers, examine failure and recovery paths, and ask whether a test could distinguish the defect. Those steps involve judgment. They belong in a focused procedure rather than a string of unrelated commands. A skill can make the procedure available without requiring the full explanation in every prompt.

Contrast that with compiling a known source list or rejecting malformed fixture rows. Those tasks need deterministic behavior. The maintained scripts and parser are better places for them. A skill may call a wrapper when appropriate, but it should not become a second, subtly different build system. Keep the executable operation in one maintained location and describe its role in the procedure.

Your toolkit also needs subtraction. Remove a redundant rule when its detail already has a current home. Replace an outdated command rather than preserving both and hoping the agent chooses correctly. Keep the next task and active evidence in the handoff, not in a growing catalogue of abandoned ideas. A file earns its place when a person can explain what repeated problem it solves.

Separate expectations from procedures

The root AGENTS.md describes recurring project expectations: consult current context, keep work bounded, use relevant checks, and report results honestly. The hardware document contains exact source-dependent facts. The review skill describes how to inspect a bounded change. The task prompt identifies the particular diff and the requested outcome. These layers should cooperate without repeating every detail.

A rule saying “run all possible checks before any response” can obstruct an inspection-only task or a harmless wording change. A better procedure connects checks to findings and scope. Reviewing a supplied diff does not automatically authorize rewriting it. If execution is available and relevant, use it to resolve a concrete uncertainty. If execution is unavailable, state the predicted consequence and the missing verification rather than inventing an observed failure.

The official [OpenAI skills guide](#) describes Codex's skill packaging and invocation; the [AGENTS.md guide](#) covers project guidance discovery. The course's installed-host behavior still needs its own fresh-session record. A valid skill file or a readable instruction summary is not evidence that the intended workflow occurred.

The firmware-review skill

Open `.agents/skills/firmware-review/SKILL.md` in CP08. Its bounded purpose is to review an embedded firmware or shared C++ policy diff for actionable correctness issues. It excludes unrelated documentation changes and full security audits. The distinction keeps a small review from becoming an uncontrolled investigation of the entire project.

The inputs are the exact diff, nearby source needed to understand it, applicable requirements, and relevant evidence. Ask the reviewer to identify which checkpoint is active before applying a checklist. CP08 does not contain the capstone command parser, so a review that demands its tests is working from the wrong stage. Figure SS16-01 shows the actual skill file and its boundaries.

The screenshot shows a VS Code workspace named 'm07-104-defect (Workspace)'. The Explorer sidebar on the left displays a file tree with the following structure:

- m07-104-defect (Workspace)
 - sensor-monitor
 - .agents \ skills
 - datasheet-to-constraints
 - firmware-review
 - SKILL.md (selected)
 - docs
 - capture-review-results.md
 - evidence.md
 - fault-exercise.md
 - handoff.md
 - hardware.md
 - interface-plan.md
 - project-brief.md
 - requirements.md
 - setup.md
 - skills.md
 - firmware
 - fixtures
 - host
 - scripts
 - tests
 - .gitignore
 - AGENTS.md
 - CHECKPOINT.md
 - dependencies.json
 - README.md

The main editor area shows the content of 'SKILL.md' with the following text:

```
1 |--
2 name: firmware-review
3 description: Use when asked to review a bounded embedded firmware or shared C++ policy diff for actionable correctness
  issues. Does not apply to unrelated documentation edits or full security audits.
4 ---
5
6 # Firmware review
7
8 Read the requested diff, its nearby callers and the applicable requirements. Keep the review within that task. Inspect
  timing arithmetic and rollover, invalid/stale/recovery paths, command and buffer bounds, integer conversions, and
  whether tests distinguish the required boundaries. Use the actual shared source; mocks cannot establish physical timing
  or electrical behavior.
9
10 Report actionable findings in order of impact, each with file/line, concrete trigger, observed or predicted
   consequence, and a minimal test or source reference. Distinguish executed evidence from a hypothesis. If no actionable
   issue is supported, say so and list only material verification gaps. Do not manufacture findings to fill a format or
   rewrite a clean diff.
11
12 Use `scripts/test.cmd` and `scripts/build-firmware.cmd` from the repository root when execution is available and within
   the task; record command exits and source identity. The host appendix uses `python3 scripts/native.py test`. A compile
   does not establish upload, serial behavior, accuracy or bench verification. Reviewing does not by itself authorize
   flashing, physical operations or remote publication.
13
```

The status bar at the bottom indicates 'Ln 1, Col 1 Spaces: 2 UTF-8 LF Skill'.

SS16-01. The actual firmware-review skill states its trigger, bounded procedure and evidence requirements.

A useful finding names a file and location, a concrete trigger, the resulting consequence, and a check or source that supports the claim. “There may be timing problems” gives the engineer little to do. “After the last valid sample ages beyond 3000 milliseconds, the changed polling path leaves the cached reading current” identifies an observable behavior and a discriminating test.

Review timing arithmetic, missing and nonfinite inputs, stale data, recovery, integer conversions, bounded work, output formatting, and relevant tests. These are inspection lenses, not a requirement to manufacture one finding in every category. If the diff is clean within the inspected scope, say that no actionable issue was supported and identify only material verification gaps.

Separate severity from certainty. A potentially serious consequence can still be a hypothesis. An executed reproducer strengthens confidence that a specific defect exists. The report should make the difference visible. Do not use dramatic wording to compensate for missing evidence, and do not downgrade an established correctness defect merely because its patch is one line.

Worked example: review the stale-data fault

Keep CP08 active in a checkout cloned from the history bundle so the review skill is available. Supply the bounded diff using `git diff CP07 FAULT-stale --`

`firmware/sensor-monitor/monitor_policy.cpp`. The FAULT-stale snapshot is CP07-based and lacks the second skill, so do not assume it discovers firmware-review on its own. The intentional mutation disables stale expiration in `poll` while still compiling. Ask for review without repair, preserving the actual response and tool evidence. Execute the faulty source only in a separate FAULT-stale exercise copy.

The intended requirement is that input age exactly 3000 milliseconds remains current, while age 3001 milliseconds becomes stale. In the supplied stale-recovery sequence, the last valid sample is at timestamp 1000. At 4000 it is still current. At 4001 its temperature must no longer be presented as current. The requirement therefore supplies a precise trigger independently of the implementation.

The expected review finding concerns the missing expiration path and its effect on state and output. It should connect `poll` to FAULT/stale and unavailable temperature, rather than merely complain about formatting. If the agent reports a finding, inspect the source and requirement yourself. The example defines the acceptance criterion; it is not a transcript of a run that you have not performed.

Run the relevant commands from the separate FAULT-stale exercise root when you are ready to verify the faulty behavior:

```
.\scripts\test.cmd
.\scripts\replay.cmd fixtures/stale-recovery.csv
```

A behaviorally useful fault exercise compiles and then fails the relevant assertion or expected sequence. A syntax error that prevents compilation does not show that the behavioral test

detects stale data. Preserve the failing output, identify the assertion, and compare it with the expected condition before attributing a cause.

After review, use a separate authorized repair task or make the correction yourself. Restore the expiration guard, repeat the same checks, and inspect the diff. A firmware/shared-source change also needs the cross-build:

```
.\scripts\build-firmware.cmd
```

The final record should distinguish review, repair, host verification, and firmware compilation. A successful result supports those software claims for the recorded source. It does not establish a physical sensor, upload, electrical behavior, or device timing. Figure SS16-02 should show the actual actionable review result and its evidence relationship.

```
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-104-defect\sensor-monitor> $ErrorActionPreference=
'Stop';$p='C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-104-defect\sensor-monitor';$q='C:\Users\us
er\Documents\AI-Embedded-Book\capture-plans-m07-m08-v01\commands\M07-L04-defect-review.ps1';if((Get-FileHash $q -Algorithm SHA2
56).Hash -ine 'e69a9dae504482b2473ef8dc0cc041e9bb097ad4278b8d03991d4bbe691261bb'){throw 'Command mismatch'};&([scriptblock]::Cr
eate([IO.File]::ReadAllText($q)))
codex-cli 0.154.0-alpha.6.2
```

Algorithm	Hash	Path
SHA256	296634BBD8167A12170C6111A68E87E70E189557DC1E651D902094888BE7175E	C:\Users\user\.vscode\extensions\ope...

- **High â€” stale samples never expire** â€” ['monitor_policy.cpp:13'](C:/Users/user/Documents/AI-Embedded-Book/capture-practi
ce-m07-m08-v01/m07-104-defect/sensor-monitor/firmware/sensor-monitor/monitor_policy.cpp:13), changed in 'FAULT-stale' to add '&
& false'.

****Trigger:**** A current sample becomes older than 3000 ms and 'poll()' runs, including the 'poll(now)' call at the start of 's
ample_at()'.

****Consequence:**** 'fail(Error::Stale)' is unreachable. The monitor can continue reporting an old temperature as current and re
main 'NORMAL'. A gap exceeding 3000 ms also fails to reset recovery, so two valid samples separated by a stale gap can incorrec
tly recover to 'NORMAL'.

****Evidence:**** ['requirements.md'](C:/Users/user/Documents/AI-Embedded-Book/capture-practice-m07-m08-v01/m07-104-defect/sensor
-monitor/docs/requirements.md) requires age greater than 3000 ms to fault, null the temperature, and reset recovery. Source ins
pection predicts failures of the existing checks at ['test_monitor.cpp:25'](C:/Users/user/Documents/AI-Embedded-Book/capture-pr
actice-m07-m08-v01/m07-104-defect/sensor-monitor/tests/test_monitor.cpp:25), ['test_monitor.cpp:43'](C:/Users/user/Documents/AI
-Embedded-Book/capture-practice-m07-m08-v01/m07-104-defect/sensor-monitor/tests/test_monitor.cpp:43), and ['test_monitor.cpp:54
'](C:/Users/user/Documents/AI-Embedded-Book/capture-practice-m07-m08-v01/m07-104-defect/sensor-monitor/tests/test_monitor.cpp:5
4). Directly stale timestamped inputs remain rejected by the separate check at line 20.

No files were changed, no ref was checked out, and CP08 remained active. I did not run builds, tests, or replay commands; all c
onsequences above are source predictions from the requested diff and nearby code.

actual_agent_exit=0

```
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-104-defect\sensor-monitor>
```

SS16-02. Actual read-only review identifies disabled stale expiry, gives trigger and consequence, and distinguishes source predictions from execution.

Test useful restraint

The defect case is only part of the skill evaluation. Create a clean comparison in a disposable CP08 copy by consistently renaming a local variable without changing behavior. Ask for a bounded review. The desired result is not a compulsory warning; it is an accurate assessment of the actual diff and any relevant limitations. Verify that the rename is genuinely consistent before calling the case clean.

Then use a fresh task to improve one README sentence without changing firmware. Do not name the review skill in this negative case. The task should remain ordinary documentation work rather than forcing a firmware audit. Explicitly invoking the skill would change the question and weaken the test of unwanted implicit selection.

Record selection separately from output quality. If the interface reveals that the skill file was read, preserve that evidence. If it does not, avoid inferring activation merely because the response uses a familiar format. A well-formed finding and an observable skill invocation answer different questions. Both can matter when maintaining the procedure after a host update.

Change one demonstrated weakness at a time. A broad description may need a tighter trigger. An unsupported finding may need a clearer evidence requirement. A clean change that provokes invented defects may need an explicit “no supported issue” outcome. Repeat the affected cases and preserve the before/after records rather than replacing failures with the final successful answer.

Write a handoff that can be checked

A handoff is a compact map of active state. It should identify the current checkpoint or commit, the completed change, the last relevant checks with evidence pointers, open questions, and one next bounded action. It should tell the next worker what can be trusted and what still needs inspection. A transcript of the entire conversation usually obscures that information.

Do not write “everything passes” when the evidence concerns an earlier source revision. Link a result to the source it checked. If the working tree changed afterward, identify which checks are still relevant and which need repetition. A saved command is useful only when its input, environment, and result can be related to the current task.

The CP08 handoff includes a concrete exercise: add an assertion proving that negative infinity produces FAULT/nonfinite with no current temperature, leaving production behavior unchanged. If you already added negative infinity in Chapter 15, use a fresh CP08 copy to avoid duplicating that extension. This is a good restart task because the requirement already covers nonfinite input, the test file contains nearby NaN and positive-infinity cases, and success can be inspected in a small diff.

Use a separate Monitor instance or carefully ordered injected timestamps so the new case does not accidentally depend on unrelated test state. For example, an independent monitor can receive negative infinity at timestamp zero. Assert its state, error category, and current flag.

The existing output tests already cover unavailable temperature formatting; distinguish a policy assertion from a JSON formatting assertion.

The expected result is a stronger test for behavior that already exists. It is not an excuse to add capstone thresholds or redesign the policy. If the new assertion fails, diagnose the discrepancy against the requirement before changing production code. The task may reveal a defect, but that finding needs evidence rather than an assumption that tests are always correct.

LAB16: resume from repository context

Allow 50-75 minutes, excluding account or dependency waiting. Extract CP08 into a new folder or create a new branch from the supplied history bundle. Preserve earlier work. Open the actual project root and inspect the starting status before creating the handoff test.

1. Read docs/handoff.md, AGENTS.md, the README, stage requirements, and the relevant test file. Confirm that negative infinity is the intended bounded next task.
2. Start a fresh agent session in this folder. Ask it to complete the handoff using repository context, without pasting the solution or commands into the prompt.
3. Observe which files it inspects, what it changes, and which maintained commands it actually runs. Preserve the exact prompt and available tool/results evidence.
4. Independently inspect the new assertion. Confirm that the expected state, error, and current flag follow from the nonfinite-input requirement and that the case does not depend on an accidental timestamp order.
5. Inspect the actual test result and any relevant replay/build result. Record source identity and limitations; an announced check is not execution.
6. Update the handoff with the completed bounded change, evidence pointers, remaining questions, and one next action. Do not claim the capstone is implemented in CP08.

Submit the review skill's defect/clean/unrelated records, the fresh-session handoff record, the small test diff, and the revised handoff. A blocked command leaves an execution gate open; the prepared documentation and static review can still be complete. Physical hardware is not a prerequisite.

Failure and recovery: a handoff without receipts

Suppose the handoff says that basic tests passed but contains no command, source identity, or saved output. The next session should not repeat that claim as established history. Search the project's evidence location and relevant prior result records. If the receipt cannot be found, mark the claim unestablished and run the applicable check when possible.

Do not overwrite the old handoff silently. Record the correction: the previous claim lacked an accessible receipt, and the current record now contains a specific result or an open dependency. This preserves the reason for the change and prevents a future reader from confusing reconstructed evidence with the original run.

Another failure is resuming the wrong checkpoint. A task that starts implementing commands in CP08 has skipped the capstone's requirements and review. Check the active source, stage README, and handoff before editing. Recover in a fresh exercise folder if necessary, preserving any useful work. Figure SS16-04 should demonstrate the actual resumed task and validation path rather than a prepared success message.

```

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-105\sensor-monitor> $ErrorActionPreference='Stop';
$p='C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-105\sensor-monitor';$q='C:\Users\user\Documents\A
I-Embedded-Book\capture-plans-m07-m08-v01\commands\M07-L05-fresh-task.ps1';if((Get-FileHash $q -Algorithm SHA256).Hash -ine '4b
c6d576d08f66df5bc4bb35c15ae59b86dabb900be94e5d1d9c6ad70a0b1e33'){throw 'Command mismatch'};&([scriptblock]::Create([IO.File]::R
eadAllText($q)))
codex-cli 0.154.0-alpha.6.2

Algorithm      Hash
-----
SHA256         296634BBD8167A12170C6111A68E87E70E189557DC1E651D902094888BE7175E      C:\Users\user\.vscode\extensions\ope...
Implemented the bounded CP08 task.

- Added a negative-infinity test proving transition from NORMAL to FAULT with `Nonfinite` and no current temperature.
- Left production firmware and policy unchanged.
- Added verification evidence to [docs/evidence.md](C:/Users/user/Documents/AI-Embedded-Book/capture-practice-m07-m08-v01/m07-1
05/sensor-monitor/docs/evidence.md).
- `.\scripts\test.cmd` **PASS**, 40 assertions, 0 failures.
- `.\scripts\replay.cmd fixtures/normal.csv` **exit 0**, expected state sequence and nonfinite fault.
- `git diff --check` **exit 0**.
- Only [tests/test_monitor.cpp](C:/Users/user/Documents/AI-Embedded-Book/capture-practice-m07-m08-v01/m07-105/sensor-monitor/te
sts/test_monitor.cpp) and the evidence note changed.

Firmware compilation was not required because no firmware or shared policy source changed. No board behavior was claimed.
actual_agent_exit=0

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m07-105\sensor-monitor>

```

SS16-04. The fresh repository-guided task reports its bounded negative-infinity change and actual validation. Independent verification also passed 40 assertions.

Maintain a small behavior baseline

After changing a skill description, relevant instructions, or the host interface, rerun a representative set: one meaningful defect, one clean diff, one unrelated task, and the repository-only handoff. Keep source and prompt inputs stable enough to compare outcomes. If both the input and procedure change, record both instead of attributing improvement to the skill alone.

Do not measure quality by the number of findings, files, or tokens. A review that correctly leaves a clean change alone can be better than a long report of speculative defects. A short handoff with valid receipts can be better than a detailed story that omits the active source. Measure whether the toolkit reduces repeated work while preserving the engineer's ability to verify the result.

Completion check and review answers

Complete the chapter when the second skill has actual defect, clean, and unrelated case records; the fresh session resumes the bounded CP08 task; and the handoff links to real current evidence. A static skill check and an instruction summary do not satisfy those runtime gates.

Why test a clean diff? A reviewer must avoid inventing correctness findings. Correct restraint is part of the procedure's value.

Why use a separate monitor for the new nonfinite case? It isolates the requirement and avoids accidental dependence on previous recovery state or timestamp order.

Does a failing compilation prove the stale-data test works? No. The useful seeded defect should compile and then be rejected for the intended behavior.

What should a fresh session do with an unsupported passing claim? Find the receipt or mark the claim unestablished and run the relevant check. It should not copy the claim into a new report.

What belongs in the handoff? Active source/state, completed work, relevant receipts, open questions, and one bounded next task. Detailed reference material stays in its own current document.

Transfer to your own work

Choose one procedure you repeat and one task you often have to reconstruct after a break. Package the procedure with a bounded output and write the task's handoff. Test them on a small real change before adding more machinery. Your toolkit is successful when another session can find the next action and verify it without relying on the old conversation.

Sources and figures

Project paths and behavior refer to the matching CP08/FAULT-stale release and its requirements. Host-specific references: [OpenAI build skills](#), [OpenAI AGENTS.md](#), and the [Agent Skills specification](#), checked September 12, 2026. Figures: SS16-01 review skill, SS16-02 actionable defect, SS16-03 handoff, SS16-04 fresh-session resume. The tests described here require actual recorded execution before a passing claim.

```

1  # CP08 review cases - actual execution record
80 ## Unrelated fresh request - README only
96
97 Coordinator compared before/after source inventories: only README.md changed.
98 Saved actual tool trace includes successful git diff --check and the final bounded diff.
99 The sentence now explicitly separates timestamped synthetic samples from physical sensor readings.
100 No firmware-review skill-file read is visible in this trace; hidden selection is unknown.
101
102 ## Handoff for a fresh session
103 Keep the three cases and runtime receipts intact; do not merge the deliberate fault into CP08.
104 Start the next task from a separate untouched CP08 copy and read AGENTS.md plus docs/handoff.md.
105 Next bounded task: negative-infinity assertions for FAULT, Nonfinite and current false using a separate Monitor.
106 Preserve production behavior. Let the new session inspect its own context and leave current verification
107 evidence.
108 Capstone thresholds, commands and hysteresis remain future work.
109
110 ## Limits and correction decision
111 The two reviews obeyed the read-only boundary; no skill repair is justified by these observations.
112 If a later case reveals a concrete skill weakness, repair that issue and repeat only the affected case.
113 These are agent/coordinator checks, not human learner acceptance or hardware measurements.
114 All physical wiring, timing and accuracy procedures remain unperformed.
115
116 ## Receipt locations in the book environment
117 `capture-evidence-m07-m08-v01/m07-l04-defect/M07-L04-defect-review/`
118 `capture-evidence-m07-m08-v01/m07-l04-clean/M07-L04-clean-review/`
119 `capture-evidence-m07-m08-v01/m07-l04-unrelated/M07-L04-unrelated/`
120 `capture-evidence-m07-m08-v01/m07-l04-runtime/M07-L04-fault-tests/`
121 `capture-evidence-m07-m08-v01/m07-l04-runtime/M07-L04-fault-stale/`
122 Each task preserves its submitted prompt, event trace, answer, exit and source identities.
123

```

SS16-03. The instructor handoff preserves case results and receipt locations, identifies limits and gives the next bounded fresh-session task.

17. Capstone: Ship a Documented Sensor Monitor

What you will be able to do

You will carry configurable alert behavior from requirements through implementation, review, verification, and a private release candidate. Start from CP08, whose normal/fault policy and basic tests already work. CP09 is the instructor reference solution and final behavior specification. Your submission includes the source, reviewed history, tests, synthetic replay, firmware-build evidence, documentation, and handoff. Physical equipment and public publication are optional and are not required to complete the core capstone.

Begin with a boundary you can defend

The feature sounds simple: alert when the temperature is high and allow the threshold to change. The engineering work lies in the details. What happens at equality? Can an invalid command alter the previous setting? Does a fault recover into an old alert state? Does reset preserve the setting? A useful requirement answers these questions before an agent generates a plausible implementation.

Keep the existing fault behavior. Missing, nonfinite, or stale input enters FAULT and invalidates the current reading. Two consecutive fresh valid samples recover; an invalid input or stale gap resets that count. Startup follows the same two-sample recovery rule. The first valid reading is current but still FAULT/recovering. Alert logic must not silently bypass this behavior.

The default threshold is 28.0 °C. While NORMAL, a valid temperature at or above the threshold enters ALERT. While ALERT, a temperature at or below threshold minus 1.0 °C clears to NORMAL. Between those boundaries, retain the prior non-fault state. Following FAULT recovery, classify from the fresh entry threshold rather than resurrecting a remembered alert.

These are teaching values, not sensor calibration or safety specifications. At the default threshold, 27.5 °C can legitimately be NORMAL or ALERT depending on the previous state. That history dependence is hysteresis. It reduces repeated transitions near one boundary, but the lesson does not claim measured noise performance or a physical control system.

Turn the contract into expected cases

Write expected states before editing the implementation. Use a small table that includes the starting condition and the input sequence:

Starting condition	Input or action	Expected outcome
Startup FAULT	25.0, then 26.0 at fresh ordered times	FAULT/recovering, then NORMAL
NORMAL, threshold 28.0	28.0	ALERT

Starting condition	Input or action	Expected outcome
ALERT, threshold 28.0	27.5	Remain ALERT
ALERT, threshold 28.0	27.0	NORMAL
NORMAL, threshold 28.0	27.5	Remain NORMAL
FAULT	Two fresh 27.5 readings	Recover to NORMAL
FAULT	Valid, invalid, valid	Remain FAULT; count restarted

Add command expectations. Accept threshold <decimal> terminated by line feed. The allowed threshold is 0-50 °C inclusive and is stored only in RAM. Maximum line length is 64 bytes before the line feed, including an optional final carriage return. A malformed, oversized, truncated, or out-of-range command must leave the prior configuration intact.

Define numeric grammar instead of accepting whatever a conversion library happens to parse. The reference accepts an optional sign, at least one integer digit, and an optional decimal point followed by at least one fractional digit. It rejects exponents, extra whitespace, trailing text, and nonfinite words. Thus threshold 30.5 can be accepted, while threshold 28junk and threshold 2e1 are malformed.

A valid command immediately re-evaluates a current non-faulted reading against the new entry threshold. It cannot clear FAULT or advance recovery. Poll the monitor's injected time before applying a command so stale data cannot be treated as current. Reset restores the default threshold and FAULT/missing; no nonvolatile persistence is promised.

Establish a preserved baseline and a small plan

Create a new exercise branch from CP08 using the supplied history bundle, or initialize your own preserved exercise repository if starting from its ZIP. A ZIP provides files; the bundle provides the teaching history. Inspect status before making changes so unrelated work is not silently included.

```
git clone .\sensor-monitor-history.bundle capstone-work
Set-Location .\capstone-work
git switch -c capstone-alerts CP08
.\scripts\test.cmd
.\scripts\replay.cmd fixtures/normal.csv
```

These baseline checks establish the stage you are extending. CP08 has no ALERT state or command parser. Its normal fixture therefore has a different expected state sequence from CP09. Record the stage with the result; do not compare output against the final reference without noticing that the implementation is earlier.

Plan four small changes: policy/configuration, command parser, adapter/output integration, and expanded verification/documentation. Keep existing shared-source architecture. The host and

firmware should compile the same policy, parser, and output implementation files rather than separate copies that can drift. Figure SS17-01 identifies the requirements and bounded plan.

requirements.md - sensor-monitor-candidate05 - Visual Studio Code

Restricted Mode is intended for safe code browsing. Trust this folder to enable all features. [Manage](#) [Learn More](#)

docs > requirements.md

R03	JSON Lines with stable sample fields and JSON null for unavailable temperature	Independent JSON parser and expected event sequences
R04	NORMAL enters ALERT at temperature $\geq$ threshold; default 28.0 °C	28.0 equality and boundary jitter
R05	ALERT clears at temperature $\leq$ threshold-1.0 °C; retain state inside band	27.0 equality; 27.5 from each direction
R06	Missing, nonfinite, or input age >3000 ms enters FAULT; stale value never current	Missing/NaN/infinity/stale tests and replay
R07	Two consecutive fresh valid samples recover; invalid input or a stale gap resets count	Recovery and interrupted recovery; after fault classify using fresh entry threshold
R08	<code>threshold <decimal></code> terminated by LF; 64-byte maximum before LF, including optional CR; 0-50 °C inclusive; RAM only	Full-line grammar, range, CRLF, NUL, overflow drain, truncation and repeated-command tests
R09	New process or <code>Monitor::reset()</code> restores threshold 28.0 and FAULT/missing	Reset tests and replay reset event

Restricted Mode

Screen Reader Optimized

SS17-01. The final CP09 reference defines inclusive alert boundaries, stale-input faults, two-sample recovery, bounded commands, and reset behavior.

CP08's native source list has no parser file. When you add `command_parser.cpp`, add it to `REFERENCE_SOURCES` in `scripts/common.cmd` so both Windows replay and test wrappers compile it. Update the source list in `scripts/native.py` if you also maintain the host appendix path. Keep the parser header and source inside `firmware/sensor-monitor`, where the Arduino sketch build includes them. Review these build-list changes with the feature; an unresolved parser symbol is an integration problem, not a reason to duplicate the parser inside a test.

Ask the agent to implement one slice at a time. Include the applicable requirement and acceptance criteria, but inspect the resulting diff before proceeding. A request to finish the whole capstone can produce a large patch whose assumptions are difficult to review. Small changes make it easier to connect a defect to the decision that introduced it.

Worked example: implement alert-entry equality

Begin with the state enum and configuration in the shared policy. Add `ALERT`, a default threshold, and the 1.0 °C hysteresis constant. Preserve the existing invalid-input checks and recovery gate. The alert comparison belongs after input validity and recovery handling, not before them.

The reference policy's relevant branches are:

```
if (state_ == State::Fault) {
    if (++recovery_count_ < 2) {
        error_ = Error::Recovering;
        return;
    }
    state_ = temperature >= threshold_ ? State::Alert : State::Normal;
} else if (state_ == State::Alert) {
    if (temperature <= threshold_ - kHysteresis)
        state_ = State::Normal;
} else if (temperature >= threshold_) {
    state_ = State::Alert;
}
```

Read the control flow before copying it. The first branch completes fault recovery and classifies the fresh temperature. The second handles clearing an existing alert. The last handles entry from `NORMAL`. Changing branch order or flattening these into one comparison can destroy hysteresis or make startup incorrectly leave `FAULT` after one sample.

For the instructor's demonstrated requirement, establish `NORMAL` with two fresh readings below the threshold, then supply exactly 28.0. Assert `ALERT`. Add a companion case immediately below the threshold. The expected outcomes come from the contract, not from the current comparison operator. Review this slice and run the applicable tests before adding parser work.

The learner then implements clearing equality, both directions through the hysteresis band, boundary jitter, and fresh fault recovery. Use ordered injected timestamps and keep each test's starting state explicit. A test at 27.5 without a known prior state is incomplete because the requirement permits two different outcomes there.

Add configuration without weakening input validation

The setter must reject nonfinite and out-of-range values without modifying the stored threshold. After a valid setting, re-evaluate a current non-faulted reading using the entry threshold. Keep FAULT unchanged. A command acknowledgement is a configuration result, not proof that a sample is valid or that recovery has occurred.

Build the parser as a bounded line collector plus a complete-line validator. Reserve storage for the maximum permitted bytes and the terminator needed by conversion. Once input exceeds the limit, drain to the next line feed and return `too_long`; do not let the remainder of the oversized line become a new command. Reset collector state at the line boundary so a subsequent valid command can be processed.

Check the entire numeric token before conversion. A conversion routine may accept the numeric prefix of 28junk; that is not the grammar this project promises. Validate in the wider numeric type before narrowing, so a value slightly above 50 cannot round into the allowed float range. The reference's complete parser is in CP09 for review after your attempt.

A partial line at host end-of-input returns `truncated`. Firmware waits for line feed without blocking the main loop; it does not invent a timeout or apply an incomplete command. The firmware limits serial work per loop so acquisition still gets a turn. These software design choices require code and tests; they do not establish observed physical serial timing.

Integrate output and replay

Add the threshold to the startup/configuration output and ALERT to state formatting. Keep the synthetic label in every relevant record. Missing, nonfinite, and stale readings produce JSON `null` for temperature. A recovering sample can contain a fresh numeric value while its state remains FAULT/recovering. Do not collapse all FAULT records into one misleading data-validity rule.

The full replay adapter supports sample, tick, command, truncated, and reset events. CP08 already supplies the normal, stale-recovery, and rollover fixtures, but it does not contain `fixtures/commands.csv`. After your initial parser and integration attempt, extract CP09 into a separate reference folder and inspect its `docs/formats.md` and `fixtures/commands.csv`. Copy only that command fixture into your exercise's `fixtures` folder, or author equivalent cases against the same contract; preserve your implementation. Use it for configuration cases and `fixtures/normal.csv` for alert transitions. Fixture syntax errors are different from intentionally invalid sensor samples. A malformed fixture exits 2 with a line-number diagnostic; a valid fixture that intentionally produces FAULT can complete with exit 0.

```
.\scripts\replay.cmd fixtures/normal.csv
.\scripts\replay.cmd fixtures/commands.csv
.\scripts\replay.cmd fixtures/stale-recovery.csv
.\scripts\replay.cmd fixtures/rollover.csv
```

At CP09, the normal fixture's expected states are FAULT, NORMAL, ALERT, ALERT, NORMAL, FAULT, FAULT, NORMAL. The first and seventh entries are recovering. This is an expected

sequence derived from the fixture and requirements; save your actual output separately. Figure SS17-04 should show a genuine synthetic replay, with no suggestion of board measurement.

The screenshot shows a Visual Studio Code workspace named 'm08-capstone (Workspace)'. The Explorer panel on the left displays the project structure, including files like 'capstone-plan.md', 'evidence.md', 'handoff.md', and 'sensor-monitor'. The main editor window shows the 'monitor_policy.cpp' file, which contains a list of synthetic test cases. The Terminal panel at the bottom shows the output of the program, which is a JSON array of test results.

```

Path
----
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m08-capstone\sensor-monitor
{"type": "startup", "version": "1.0.0", "synthetic": true, "input_status": "missing", "threshold_c": 28.00, "hysteresis_c": 1.00}
{"type": "sample", "synthetic": true, "uptime_ms": 0, "temperature_c": 25.00, "state": "FAULT", "error": "recovering"}
{"type": "sample", "synthetic": true, "uptime_ms": 1000, "temperature_c": 26.00, "state": "NORMAL", "error": "none"}
{"type": "sample", "synthetic": true, "uptime_ms": 2000, "temperature_c": 28.00, "state": "ALERT", "error": "none"}
{"type": "sample", "synthetic": true, "uptime_ms": 3000, "temperature_c": 27.50, "state": "ALERT", "error": "none"}
{"type": "sample", "synthetic": true, "uptime_ms": 4000, "temperature_c": 27.00, "state": "NORMAL", "error": "none"}
{"type": "sample", "synthetic": true, "uptime_ms": 5000, "temperature_c": null, "state": "FAULT", "error": "nonfinite"}
{"type": "sample", "synthetic": true, "uptime_ms": 6000, "temperature_c": 25.00, "state": "FAULT", "error": "recovering"}
{"type": "sample", "synthetic": true, "uptime_ms": 7000, "temperature_c": 25.00, "state": "NORMAL", "error": "none"}
actual_exit=0
CANDIDATE commands
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m08-capstone\sensor-monitor
{"type": "startup", "version": "1.0.0", "synthetic": true, "input_status": "missing", "threshold_c": 28.00, "hysteresis_c": 1.00}
{"type": "sample", "synthetic": true, "uptime_ms": 0, "temperature_c": 28.00, "state": "FAULT", "error": "recovering"}
{"type": "sample", "synthetic": true, "uptime_ms": 1000, "temperature_c": 28.00, "state": "ALERT", "error": "none"}
{"type": "command", "synthetic": true, "command": "threshold", "result": "accepted", "threshold_c": 30.00}
{"type": "sample", "synthetic": true, "uptime_ms": 1000, "temperature_c": 28.00, "state": "NORMAL", "error": "none"}
{"type": "command", "synthetic": true, "command": "threshold", "result": "malformed", "threshold_c": 30.00}
{"type": "sample", "synthetic": true, "uptime_ms": 2000, "temperature_c": 28.00, "state": "NORMAL", "error": "none"}
{"type": "command", "synthetic": true, "command": "threshold", "result": "out_of_range", "threshold_c": 30.00}
{"type": "sample", "synthetic": true, "uptime_ms": 2000, "temperature_c": 28.00, "state": "NORMAL", "error": "none"}
{"type": "command", "synthetic": true, "command": "threshold", "result": "truncated", "threshold_c": 30.00}
{"type": "sample", "synthetic": true, "uptime_ms": 2000, "temperature_c": 28.00, "state": "NORMAL", "error": "none"}
{"type": "command", "synthetic": true, "command": "threshold", "result": "accepted", "threshold_c": 28.00}
{"type": "sample", "synthetic": true, "uptime_ms": 2000, "temperature_c": 28.00, "state": "ALERT", "error": "none"}
{"type": "sample", "synthetic": true, "uptime_ms": 3000, "temperature_c": null, "state": "FAULT", "error": "missing"}
{"type": "sample", "synthetic": true, "uptime_ms": 4000, "temperature_c": 27.50, "state": "FAULT", "error": "recovering"}
{"type": "sample", "synthetic": true, "uptime_ms": 5000, "temperature_c": 27.50, "state": "NORMAL", "error": "none"}
actual_exit=0

```

SS17-04. The synthetic command replay changes threshold and state, preserves rejected settings, and resets to missing input.

Verify boundaries and meaningful rejection

Run the full host assertions, inspect their result, and cross-build the reviewed source:

```
.\scripts\test.cmd  
.\scripts\build-firmware.cmd
```

Cover inclusive entry and clearing, jitter, both directions through the band, invalid and stale input, interrupted recovery, timer rollover, valid threshold updates, malformed numeric suffixes, range endpoints, oversized lines, repeated commands, truncation, and reset defaults. The independent replay checks in the reference solution also parse JSON and compare expected sequences. State which checks you actually ran.

Use isolated FAULT-hysteresis and FAULT-command copies to demonstrate that tests reject meaningful defects. The hysteresis mutation changes inclusive clearing to strict clearing, missing exactly 27.0 °C. The command mutation accepts malformed trailing text. Both must compile before their behavioral rejection supports the intended lesson. Preserve a fresh good solution and show that it still passes.

Do not infer completeness from a large assertion count. A test suite that misses equality or validates only successful commands can look substantial while leaving the important defect undetected. Map the tests to requirements and name the limits of host execution. Figure SS17-03 shows actual outcomes, while the evidence note explains their scope.


```

No tests, builds, firmware, host programs, or remote services were run. Inspection used read-only file, diff, status, search, hash, and `git diff --check` commands; `git diff --check` exited successfully.
actual_agent_exit=0

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m08-capstone\sensor-monitor> $ErrorActionPreference='Stop';$q='C:\Users\user\Documents\AI-Embedded-Book\capture-evidence-m07-m08-v01\m08-capstone\repair-source01.ps1';if((Get-FileHash $q -Algorithm SHA256).Hash -ine '28c03b7584e45c4dfb36e2aaa81528527a15c7c7e118b1bfd2db223945b33e53'){throw 'Repair input changed'};&([scriptblock]::Create([IO.File]::ReadAllText($q)));$q='C:\Users\user\Documents\AI-Embedded-Book\capture-evidence-m07-m08-v01\m08-capstone\configuration-repro-after01.ps1';if((Get-FileHash $q -Algorithm SHA256).Hash -ine '0bfdd122edd0cf8ee47ce1f82f19b88b83606dd03f4c65d93a7fa2ef08b08133'){throw 'Repair input changed'};&([scriptblock]::Create([IO.File]::ReadAllText($q)));$q='C:\Users\user\Documents\AI-Embedded-Book\capture-plans-m07-m08-v01\commands\M08-L02-docs-tests.ps1';if((Get-FileHash $q -Algorithm SHA256).Hash -ine 'd154e6b88747890aa88d2bce24aeaa0cb1b8ab43598578d686976f898531f330'){throw 'Test wrapper changed'};$s=[IO.File]::ReadAllText($q).Replace('M08-L02-docs-tests','M08-L02-repair-tests');&([scriptblock]::Create($s))
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m08-capstone\sensor-monitor\firmware\sensor-monitor\monitor_policy.cpp SHA256=50a458fb565aef433f4d089720d0da7f571f02c4934597ad0c7ebb26f633b57f
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m08-capstone\sensor-monitor\tests\test_monitor.cpp SHA256=6494fc22d4531112b907592d463a549eb78114447181241b55f571b54525d3c4
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m08-capstone\sensor-monitor\docs\requirements.md SHA256=d16a46333e03c48400b74de4de522628c7e69e9056357f3b724ed42fe649365
Coordinator repair applied; current checks pending
setup_ALERT=1; setting_accepted=1; expected=NORMAL; actual=NORMAL
compile_exit=0; actual_repro_exit=0

Path
----
C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m08-capstone\sensor-monitor
PASS 140 assertions; 0 failures; synthetic host tests
actual_exit=0

PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m08-capstone\sensor-monitor>

```

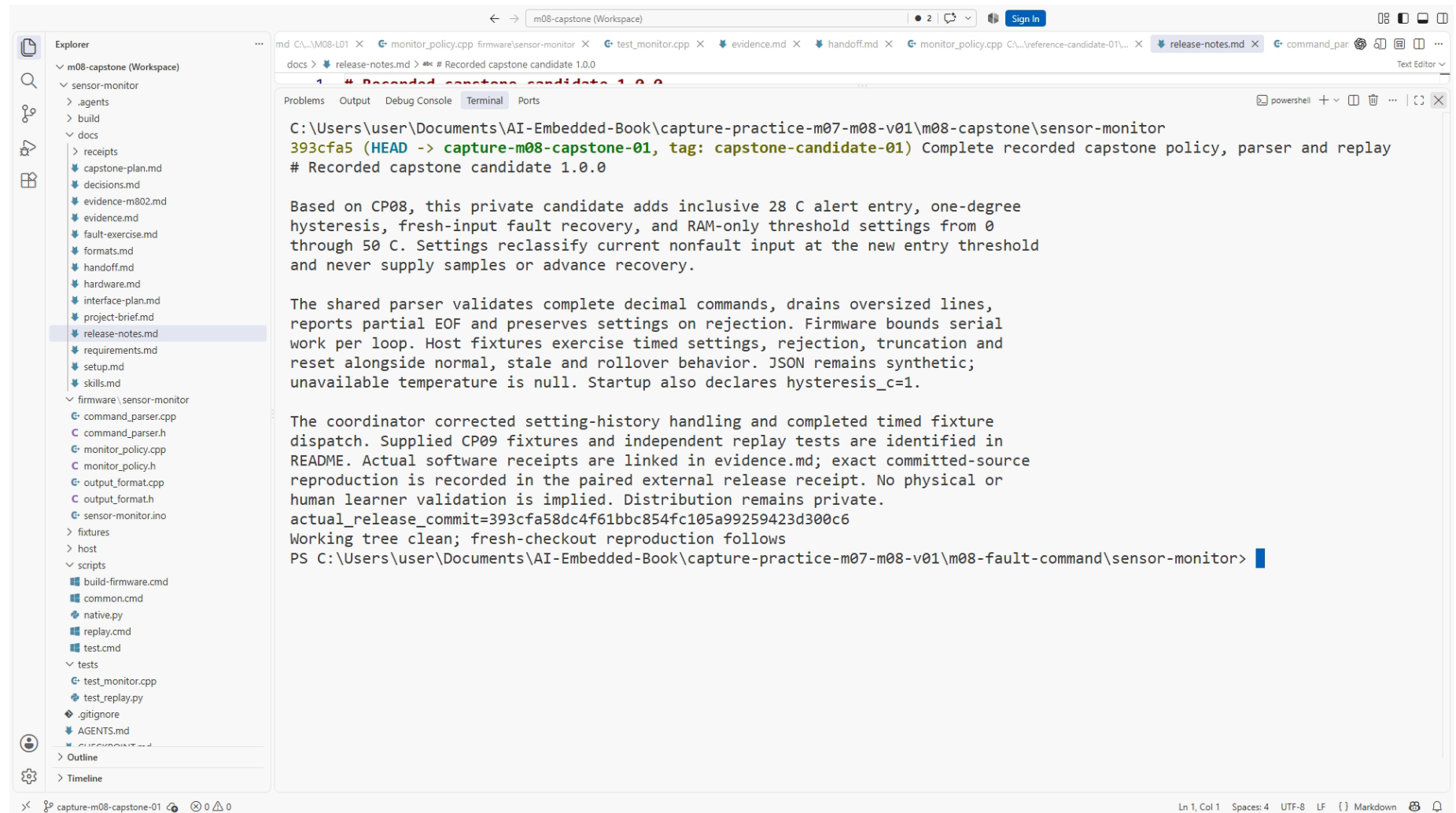
SS17-03. The corrected candidate returns NORMAL after the threshold-setting regression and passes all 140 synthetic host assertions.

Package a reproducible private candidate

Update the README, requirements, formats, decisions, evidence summary, handoff, and release notes to match the reviewed implementation. Record the release identity, source commit or hashes, exact target/tool inputs, available commands, and known limitations. Keep learner code and permitted fixtures separate from private authoring credentials, unrelated history, and production administration.

Create a fresh second checkout from the intended release source and follow the written setup. Run tests, replay, and firmware compilation there. This detects dependencies on untracked local files or incidental working-directory state. A clean status alone is not reproduction; the commands must actually run in the second copy. Preserve the receipt and distinguish it from checks in the original exercise folder.

A private release candidate satisfies the course task. Publishing a public repository, buying hardware, or uploading to a device is unnecessary. The candidate becomes reviewable through complete files and evidence, not through public visibility. Figure SS17-05 identifies release notes and source history without exposing unrelated authoring material.



```

C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m08-capstone\sensor-monitor
393cfa5 (HEAD -> capture-m08-capstone-01, tag: capstone-candidate-01) Complete recorded capstone policy, parser and replay
# Recorded capstone candidate 1.0.0

Based on CP08, this private candidate adds inclusive 28 C alert entry, one-degree
hysteresis, fresh-input fault recovery, and RAM-only threshold settings from 0
through 50 C. Settings reclassify current nonfault input at the new entry threshold
and never supply samples or advance recovery.

The shared parser validates complete decimal commands, drains oversized lines,
reports partial EOF and preserves settings on rejection. Firmware bounds serial
work per loop. Host fixtures exercise timed settings, rejection, truncation and
reset alongside normal, stale and rollover behavior. JSON remains synthetic;
unavailable temperature is null. Startup also declares hysteresis_c=1.

The coordinator corrected setting-history handling and completed timed fixture
dispatch. Supplied CP09 fixtures and independent replay tests are identified in
README. Actual software receipts are linked in evidence.md; exact committed-source
reproduction is recorded in the paired external release receipt. No physical or
human learner validation is implied. Distribution remains private.
actual_release_commit=393cfa58dc4f61bbc854fc105a99259423d300c6
Working tree clean; fresh-checkout reproduction follows
PS C:\Users\user\Documents\AI-Embedded-Book\capture-practice-m07-m08-v01\m08-fault-command\sensor-monitor>

```

SS17-05. The private candidate records its local version tag, source commit, release notes and validation limits.

LAB17: complete and defend the capstone

Allow several practice sessions. Begin from CP08, preserve the baseline, and implement the undemonstrated requirements using the plan above. Consult CP09 only after your own attempt or to diagnose a specific uncertainty. Submit the final source, bounded commits/diff, requirement-to-test map, actual host/replay/build receipts, seeded-defect evidence, fresh-checkout result, documentation, and handoff.

Use this 100-point rubric:

Category	Points	Observable evidence
Requirements and sourced hardware facts	15	Explicit boundaries, exact documented target, traceable sources and unknowns
Git history and review	15	Coherent changes and inspected diff
Firmware behavior	25	Alerts, hysteresis, parser, faults, recovery, reset
Verification quality	25	Independent expectations, meaningful defects rejected, replay and limitations
Documentation and handoff	10	Reproducible setup and current evidence pointers
Agent instructions and skills	10	Actual fresh-session and skill case records

The suggested pass is 80 points plus mandatory reproducible build, critical software behavior, host/replay checks, and honest evidence labeling. A high total cannot compensate for invented results or missing critical behavior. Physical validation has a separate optional checklist and is not required for the core score.

Failure and recovery: a correct program, an incorrect claim

Suppose the software checks pass and the release notes say “sensor hardware verified.” Correct the claim to name the actual host tests, synthetic replay, and firmware compilation. Then list the unperformed physical checks. The correction does not erase the successful software work; it prevents readers from relying on evidence that does not exist.

If a required check fails, preserve its original reproduction and narrow the next change. A parser error needs parser evidence; an electrical hypothesis cannot explain a host-only fixture failure without an additional argument. Repair the demonstrated cause, rerun the affected checks, then perform the final integrated verification after the last relevant change.

Completion check, questions, and transfer

Complete the core capstone when the mandatory behavior and evidence gates pass, the rubric reaches its target, and another checkout reproduces the result. Account or dependency limits leave affected gates open; they do not authorize a passing claim from prepared scripts.

Why can 27.5 °C have two valid states? Inside the hysteresis band, the prior non-fault state determines retention. Starting state belongs in the test.

Can a valid command clear FAULT? No. It changes configuration but does not supply the two fresh valid samples required for recovery.

Why reject 28junk? The entire command must match the promised grammar. Parsing only a numeric prefix violates that contract.

Why check a wider value before narrowing? A slightly out-of-range input must not round into the allowed range during conversion.

What does fresh-checkout reproduction add? It checks the completeness of the distributed source and setup, exposing reliance on untracked or incidental local state.

Transfer the workflow to one bounded feature in another project: explicit requirements, a small plan, independent expectations, reviewed changes, real checks, and a current handoff. Keep the evidence boundary as precise as the feature itself.

Sources and figures

The accompanying CP08 and CP09 source, requirements, formats, fixtures, and checkpoint notes define the executable examples. The documented target remains [Adafruit product 5477](#); physical behavior is not established by that reference. Figures: SS17-01 requirements, SS17-02 diff, SS17-03 tests, SS17-04 synthetic replay, SS17-05 private candidate. Optional HP06 requires a separately performed physical lab and is not part of core completion.

The screenshot shows a code editor with a diff view of `monitor_policy.cpp`. The diff highlights changes to the `sample_at` function. The changes include adding classification logic after recovery and after a valid threshold change.

```

20 @@ -23,12 +31,22 @@ void Monitor::sample_at(uint32_t now, uint32_t sampled_at, float temperature, bo
21     current_ = true;
22     if (state_ == State::Fault) {
23         if (++recovery_count_ < 2) { error_ = Error::Recovering; return; }
24         - state_ = State::Normal;
25         + classify(false);
26     } else {
27         + classify(true);
28     }
29     error_ = Error::None;
30     recovery_count_ = 0;
31 }
32
33 +bool Monitor::set_threshold(double threshold_c) {
34 + if (!std::isfinite(threshold_c) || threshold_c < kMinimumThresholdC ||
35 +     threshold_c > kMaximumThresholdC) return false;
36 + threshold_c = static_cast<float>(threshold_c);
37 + if (state_ != State::Fault && current_) classify(false);
38 + return true;
39 +}
40 +
41 Snapshot Monitor::snapshot(uint32_t now) const {
42     return {now, temperature_, state_, error_, current_};
43 }
44

```

The Explorer sidebar shows the project structure for `m08-capstone`:

- m08-capstone (Workspace)
 - sensor-monitor
 - .agents
 - build
 - docs
 - receipts
 - capstone-plan.md
 - decisions.md
 - evidence-m802.md
 - evidence.md
 - fault-exercise.md
 - formats.md
 - handoff.md
 - hardware.md
 - interface-plan.md
 - project-brief.md
 - release-notes.md
 - requirements.md
 - setup.md
 - skills.md
 - firmware\sensor-monitor
 - command_parser.cpp
 - command_parser.h
 - monitor_policy.cpp
 - monitor_policy.h
 - output_format.cpp
 - output_format.h
 - sensor-monitor.ino
 - fixtures
 - host
 - scripts
 - build-firmware.cmd
 - common.cmd
 - native.py
 - replay.cmd
 - test.cmd
 - tests
 - test_monitor.cpp
 - test_replay.py
 - .gitignore
 - AGENTS.md
 - Outline
 - Timeline

SS17-02. The bounded policy diff adds classification after recovery and after a valid threshold change.

18. Maintain the Workflow and Choose What Comes Next

What you will be able to do

You will evaluate the whole effort of an AI-assisted task, maintain the sources and tools your project relies on, and apply the workflow to one bounded change of your own. Start with the completed core capstone and its evidence bundle. Your deliverable is a personal workflow and maintenance plan supported by a real transfer exercise. You do not need a new framework, another subscription, or a different board to finish this chapter.

Measure verified delivery

The first generated answer is easy to notice. The work that follows can be less visible: understanding the patch, finding an incorrect assumption, repairing it, running checks, and updating the handoff. If you measure only generation time, you can conclude that a workflow is faster while spending more effort getting to a result you can trust.

Define the end of a task before comparing approaches. For Sensor Monitor, a small maintenance task is finished when the requested artifact is correct, its diff is reviewed, relevant checks have actual results, and the documentation states any remaining limits. A proposed patch or a confident response is an intermediate state. The end point should be the same whether you work manually, use an agent, or combine both.

Track active effort and waiting separately. Active effort includes task preparation, review, diagnosis, repair, and evidence writing. Waiting may include generation, compilation, downloads, or account limits. These categories answer different questions. A workflow can reduce active effort while increasing elapsed time, or shorten generation while increasing repair. Record both rather than hiding the tradeoff in one attractive number.

A practical worksheet has task identity, starting source, acceptance criteria, approach, preparation time, generation or editing time, review/repair time, verification time, waiting time, final result, and limitations. Record the units and measurement method. You do not need precise instrumentation for a small comparison, but you do need to avoid presenting remembered estimates as measured observations.

Compare on representative work

Choose a task you actually perform. A spectacular one-off demonstration may reveal a capability but tell you little about routine maintenance. Useful comparison tasks include clarifying an error message, adding a missing boundary case, reviewing a bounded parser diff, or improving a setup instruction. Keep the starting source and acceptance criteria stable enough that the results are comparable.

Do not give one approach a perfect brief and the other an ambiguous request. The quality of task preparation is part of the workflow, but it should be visible. If the agent needed several

rounds of clarification, count that effort. If a manual approach relied on knowledge you already had, record that advantage. The purpose is to understand your work, not to prove that one category must always win.

Output can vary across model runs and tool versions. A single run supports a limited observation, not a permanent ranking. When a decision matters, repeat representative tasks and look for recurring strengths and weaknesses. Preserve actual prompts, versions, and artifacts so a future comparison does not depend on vague recollection of which assistant seemed impressive.

Use a rubric that includes correctness, scope, evidence quality, explainability, and total effort. A fast patch that you cannot explain may impose a maintenance cost later. A longer first pass that creates a clear test or source record may reduce the next task's effort. Evaluate those consequences against your project needs rather than rewarding output volume.

Worked example: clarify two kinds of failure

Use a disposable CP09 copy and choose this maintenance task: improve the README's explanation of a malformed fixture versus an intentional invalid sensor sample. The acceptance criteria are small. The documentation must distinguish exit 2 with a fixture line diagnostic from a valid replay that intentionally reaches FAULT and can exit 0. It must preserve synthetic labeling and avoid implying physical sensor failure.

For approach A, read `docs/formats.md`, inspect the relevant fixtures, and write the paragraph yourself. For approach B, start from the same unmodified copy and ask an agent for the same bounded clarification using the supplied project sources. Do not tell the agent to change parser behavior; the task concerns explanation. Preserve both resulting paragraphs and their diffs.

Here are two original instructional candidates to evaluate, not claimed outputs from those approaches:

A malformed fixture is rejected with a line-number diagnostic and exit 2. A valid synthetic fixture can intentionally produce FAULT while the replay process exits 0.

A FAULT result means the program failed; rerun until every reading is normal.

The first candidate distinguishes input-file validity from policy state. The second misinterprets a legitimate fault scenario and encourages hiding useful evidence. The difference can be established from the project's documented format and actual reproduction, regardless of which author or tool produced the text.

Run the relevant replay commands in the preserved exercise root:

```
.\scripts\replay.cmd fixtures/invalid.csv
.\scripts\replay.cmd fixtures/normal.csv
```

The first fixture is intentionally malformed; preserve its diagnostic rather than editing it during this comparison. The second contains a nonfinite sample that legitimately produces FAULT. Read the actual process results and output. Record them as synthetic software evidence. The

expected distinction is defined here; your evidence record must contain what your own run actually did.

Score both approaches against the same criteria. Did the paragraph preserve the distinction? Did it change only the requested documentation? Can you trace its explanation to the source and actual check? How much preparation, review, correction, and verification did it take? Leave timing fields unmeasured if you did not measure them; do not supply plausible numbers to complete a table.

The exercise can reveal different outcomes. An agent may draft the correct distinction quickly but require careful scope review. A manual pass may take longer to locate the source but produce a paragraph you understand immediately. Report the result of your actual attempts. Neither outcome should be generalized beyond the available evidence.

Maintain sources as dependencies

A source link can become stale even when the code still compiles. A manufacturer may revise a board, a library may change behavior, or a tool may move a setting. Record the source version or access date where it matters, and recheck the claims that an update could affect. Updating every document on every session creates unnecessary work; ignoring relevant changes creates hidden assumptions.

The exact product identifier is more durable than a remembered family description, but it still needs a relevant source. The project's dependency record preserves the tested target and tool inputs. When those inputs change, record the reason, inspect the difference, and repeat the affected checks. A successful old build does not certify a new toolchain configuration.

Separate a documentation refresh from an implementation change. A page may move without changing a technical fact. A library update may require both source edits and new tests. A revised model interface may affect skill discovery while leaving the C++ program unchanged. Classify the change so verification addresses the affected behavior rather than treating every update as a complete project restart.

Keep a short maintenance table with the item, current recorded version or source, review trigger, relevant check, and last verified result. For the compiler, a trigger is changing its version or runtime setup. For a skill, it is changing the description, instructions, or host behavior. For a hardware fact, it is changing the exact board or encountering a source discrepancy.

Preserve a working baseline before updates

Use a separate branch or exercise copy for an update. Record the current source identity and relevant passing evidence before changing dependencies. Read the official release or installation guidance for the component you are changing. Do not silently replace the tool in every environment and then discover that the earlier reproduction path has disappeared.

After the update, run the smallest relevant set that can establish the effect. A compiler change calls for builds and behavior checks. A skill trigger change calls for positive and negative

selection cases. A wording-only link repair calls for link and rendered-content checks. Broaden testing when the change, a failure, or an unresolved concern justifies it.

If the update fails, preserve its result and return to the documented working baseline in the disposable environment. A rollback should restore a known state, not erase evidence of why the update was rejected. Record the missing capability or incompatibility and the next useful investigation. You can continue independent project work while that update remains unresolved.

Avoid maintaining multiple unexplained command paths. One tested wrapper plus clearly labeled platform notes is easier to support than several copied command lines that gradually diverge. The primary Windows course path uses `.cmd` wrappers; host appendix commands have their own evidence boundary. Do not infer Windows behavior from a macOS test merely because both compile C++.

Share deliberately

A useful release contains the source, setup, dependency information, permitted fixtures, documentation, and evidence needed by its audience. It should not accidentally contain private credentials, unrelated history, authoring service configuration, or personal screen captures. Review the actual package rather than assuming a clean repository view proves that every exported asset is appropriate to share.

Repository visibility and licensing answer different questions. A private candidate can be reviewed without a public launch. A public repository does not by itself communicate all intended reuse permissions. GitHub's [repository licensing guide](#) explains why a deliberate license record matters. Preserve existing dependency notices and distinguish original code, written material, images, and other assets when recording their rights.

Do not let an agent assign broad reuse rights to material whose origin is uncertain. Identify what you created, what came from dependencies, and what is linked rather than redistributed. A source citation supports provenance but does not automatically grant permission to copy a whole document. The release decision should reflect the material actually included.

Keep publication separate from verification. A private release can satisfy the engineering task. A public release adds audience, support, and distribution decisions. You can prepare a complete reviewable package before making those choices. The course does not require a public upload to demonstrate competence with the workflow.

Transfer without copying assumptions

When moving to another board or project, copy the workflow before copying hardware facts. Keep the pattern of exact identity, sourced constraints, explicit requirements, isolated policy, and bounded checks. Replace the target-specific details using the new project's primary material. A pin mapping, memory option, or peripheral-power alias should not travel merely because the processor family sounds familiar.

Identify which parts can remain unchanged. Hardware-independent parsing or state logic may be reusable if its requirements still apply. An acquisition adapter, timing assumption, output channel, or build target may need new verification. Document that boundary before asking the agent to port everything. A successful compilation on the new target still does not establish physical behavior.

For a software-only transfer, choose a small feature whose result you can observe without new equipment. Add a missing error-path test, improve a reproducible setup step, or clarify a report format. The same workflow applies: establish the starting state, write acceptance criteria, make the bounded change, review it, run relevant checks, and update the handoff.

If your own project contains sensitive material or actions outside the current task, keep the transfer scoped to an appropriate local copy and available tools. The useful habit is deciding what evidence and access the task needs. An instruction file cannot replace the actual permission boundary, and a reuse exercise should not quietly expand into unrelated operations.

Choose a next learning question

A real-time operating system can introduce scheduling, synchronization, and ownership questions that a simple loop does not expose. Study one of those questions in a bounded example before replacing the project's architecture. The useful outcome might be a testable queue or timeout contract, not a complete port. The official [FreeRTOS documentation](#) is a starting reference for that optional direction.

PCB assistance can apply the same source discipline to a schematic or layout review: exact components, constraints, and reviewable changes. AI interpretation of an image does not replace checking small labels, connectivity, or design rules in the actual engineering files. Begin with one limited review question and a way to verify the answer independently.

Local model experiments shift the evaluation toward the capabilities, resource needs, and data handling of a particular runtime. Do not assume that a local deployment matches a hosted model's tools or skill discovery. Compare representative tasks using the same total-effort rubric and preserve the exact configuration. No local model installation is required to finish this course.

TinyML introduces model inference on a constrained device, which is different from using an assistant to help write firmware. Google's [LiteRT for Microcontrollers overview](#) describes one supported runtime direction. A future TinyML task needs its own model, input, memory, latency, and quality checks. The present Sensor Monitor contains no on-device AI inference.

LAB18: complete one transfer task

Choose a bounded task in your own appropriate project or use the README clarification from the worked example. Write a brief naming the goal, starting source, relevant context, scope, and observable acceptance condition. Record the baseline and preserve a recovery path before editing.

Complete the task using the workflow you want to evaluate. Keep a simple effort record that separates preparation, editing/generation, review/repair, verification, and waiting. Save the actual diff and relevant results. Do not add a new tool merely to make the exercise appear more advanced.

Write a short maintenance plan with three items that matter to this task: the current source or tool version, the trigger for review, and the check that would establish continued behavior. Finish with a handoff naming the next bounded action or explicitly stating that this task is complete with its evidence.

Submit the brief, artifact, diff, results, effort record, and maintenance plan. If comparing two approaches, use separate preserved starts and common criteria. Do not invent timing measurements or human feedback. The transfer succeeds when you can explain both the result and how you know it meets the stated requirement.

Failure, recovery, and completion

The common failure is declaring the fastest generator the fastest workflow while ignoring repair and verification. Recover by returning to the same completion definition and including the omitted work. If timing was not measured, report the comparison qualitatively and plan a better measurement next time. False precision is not an improvement.

Complete this chapter when your transfer task has an observable result, actual evidence, a current handoff, and a maintenance plan tied to realistic change triggers. Finish the core course with the capstone rubric and evidence intact; optional future topics do not add hidden completion requirements.

Why count review and repair? They are part of reaching a result you can use and maintain. Generation alone is an intermediate step.

Does a tool update require every check? It requires checks relevant to what changed, broadened when failures or uncertainty justify it.

What should transfer to another board first? The workflow and evidence habits. Hardware-specific facts must be re-established for the exact new target.

Is public visibility the same as licensing? No. Distribution visibility and reuse permissions are distinct decisions, and included material needs accurate provenance.

Does this course run AI on the device? No. It teaches AI-assisted engineering. A TinyML extension would require a separate implementation and validation task.

Sources and figures

Project behavior and commands come from the matching CP09 release. Optional directions and sharing references: [GitHub licensing](#), [FreeRTOS documentation](#), and [LiteRT for Microcontrollers](#), checked September 12, 2026. Figures: SS18-01 measured comparison including review/repair, SS18-02 actual transfer task, SS18-03 maintenance plan. Illustrative paragraphs in this chapter are authored examples, not invented agent outputs or observed timing results.

```

1 # README transfer: actual effort and limits
2
3 Task: explain modeled synthetic FAULT / exit 0 versus malformed fixture / exit 2.
4 Source: separate supplied CP09 copy; only README.md changed.
5 Author: production coordinator, manual bounded edit; no AI/manual comparison.
6
7 | Stage | Active effort | Waiting | Actual observation |
8 | --- | --- | --- | --- |
9 | Preparation and source reading | Unmeasured | Unmeasured | Existing task brief, formats, fixtures and README |
10 | Editing | Unmeasured | Unmeasured | One guarded README section; original bytes retained |
11 | Review | Unmeasured | Unmeasured | Diff inspected; no parser, fixture or firmware change |
12 | Diagnosis / repair | Unmeasured | Unmeasured | No task repair needed; both expected exits observed |
13 | Verification | Unmeasured | Unmeasured | Two real replays; source maps unchanged by both |
14
15 Recorded execution window: 2026-09-14 06:38:48.333 to 06:39:13.529 UTC,
16 25.196 seconds. Includes 22 seconds of deliberate display holds. This is not
17 active effort or end-to-end delivery time; preparation and review are outside it.
18
19 Conditions: pinned compiler already installed; fresh CP09 exercise copy;
20 no preexisting build directory copied; local video assembly ran concurrently.
21 No timing fields were inferred from narration, model speed or output length.
22
23 Observation: the short wording change still required source reading, diff review,
24 two actual process results and a source-integrity comparison. Typing alone would
25 not establish a verified explanation. No human learner trial was performed.
26

```

SS18-01. The effort record separates unmeasured active work from a measured execution window and its display holds.

The screenshot shows a VS Code editor window with a workspace named 'm08-capstone (Workspace)'. The Explorer sidebar on the left shows a file tree with folders like 'sensor-monitor', 'docs', and 'firmware'. The main editor area displays a README.md file with the following content:

```

1  # Sensor Monitor
7  ## Windows quick start
22 .\scripts\replay.cmd --commands
23 ```
24
25 Enter `threshold 30` then Enter. Enter `threshold 28junk` to see rejection. Press Ctrl+Z then Enter to end Windows
terminal input. A new process restores 28.0 °C. Command mode demonstrates parser/configuration only; fixture mode
supplies sample events. On the host appendix path, run `python3 scripts/native.py test`, `python3 scripts/native.py
replay`, or `python3 scripts/native.py commands` with a C++17 compiler installed. The independent JSON replay checks
use `python3 scripts/native.py build` then `python3 tests/test_replay.py`.
26
27 ## Replay success and fixture errors
28
29 A synthetic FAULT is a modeled monitor state, not a failed replay process.
30 `normal.csv` includes recovery and invalid-sample states and completes with exit 0.
31 Run `.\scripts\replay.cmd fixtures/normal.csv` and inspect both the state sequence
32 and `$LASTEXITCODE`.
33
34 A malformed fixture is different. Run `.\scripts\replay.cmd fixtures/invalid.csv`:
35 the parser reports a fixture line 3 diagnostic and returns exit 2. Earlier valid
36 output remains visible; it does not make the malformed input successful. Check
37 [the fixture grammar](docs/formats.md) before changing a fixture. Do not change
38 the parser or firmware to turn this intentional rejection into a pass.
39
40 ## Read the project
41
42 - [Requirements and timing contract](docs/requirements.md)
43 - [Output and fixture formats](docs/formats.md)
44 - [Exact hardware reference and optional deployment](docs/hardware.md)
45 - [Design decisions](docs/decisions.md)
46 - [Checkpoint starts, recovery, and fault exercises](docs/checkpoints.md)
47 - [Evidence limits](docs/evidence.md) and [fresh-session handoff](docs/handoff.md)

```

The status bar at the bottom indicates 'Ln 32, Col 1', 'Spaces: 4', 'UTF-8', 'LF', and 'Markdown'.

SS18-02. The bounded README clarification distinguishes modeled FAULT from malformed-fixture rejection.

```

1 # Three-item maintenance plan
2
3 1. Compiler, core and target
4 Current: LLVM 23.1.1 / llvm-mingw 20260908; Arduino CLI 1.5.1;
5 ESP32 core 3.3.11. Full target remains in dependencies.json:
6 esp32:esp32:adafruit_feather_esp32s3 with CDCOnBoot=cdc, PSRAM=enabled,
7 FlashSize=4M, CPUFreq=240, FlashMode=qio, PartitionScheme=tinyuf2_noota.
8 Trigger: version or target option changes. Preserve a baseline copy and run
9 native behavior, replay and cross-build checks on the changed configuration.
10 A new board also needs its own product, pin, power and physical evidence.
11
12 2. Firmware-review skill
13 Current SHA256:
14 3e52d2ba4936cf1fe14e5c8c86e9468786466a24a0bd94c17c79d7263dcb33bd
15 Trigger: description or procedure edit. Repeat relevant defect, clean,
16 unrelated and incomplete-input cases as applicable. Preserve exact prompts
17 and observed tool reads; do not invent hidden selection evidence.
18
19 3. README setup link
20 Current destination: docs/setup.md
21 Current SHA256:
22 61a006dd7aedb92d1b6c24c28c021952f93b01b724b4fb6fd79cd39d1f42ff94
23 Trigger: link or destination changes. Open the actual link and inspect rendered
24 intended content; run only affected setup steps if the instructions changed.
25
26 Updates are not performed merely to demonstrate maintenance. Use a separate
27 copy, consult official guidance when changing a tool, and preserve failures
28 and the working baseline. No security settings or paid access need changing.
29

```

SS18-03. Three maintenance items connect recorded identities to review triggers and relevant checks.

Practical references

Use these references to resume a specific task. They keep the main chapters readable while preserving exact recovery and record formats.

Reference	Use when
A. Terminal and OS notes	A path, shell, executable, or platform difference interrupts work
B. Git recovery	You need to preserve work while changing staging, working content, or history
C. Markdown and YAML	A document, link, code fence, or skill header is malformed
D. Task-brief patterns	A request needs a clear boundary and observable result
E. Project records	You need requirements, sources, decisions, tests, or a handoff
F. Instructions and skills	A repeatable procedure needs a focused package and behavior checks
G. Versions and checkpoints	You need the supported starting state and applicable commands
H. Troubleshooting	You need to classify a failure before choosing a remedy
I. Sources, rights, and errata	A claim, asset, or correction needs a traceable record
Glossary	A term needs a short practical definition

Examples marked illustrative are original instructional material. Commands require the stated execution context. Record the output of your own run rather than copying an expected result into an evidence field.

A. Terminal and operating-system notes

A terminal displays commands and their output. A shell interprets those commands. Windows examples in this edition use PowerShell; a `.cmd` file launched from it runs a Windows command wrapper. The working directory determines what a relative path identifies. Confirm that directory before diagnosing a missing file or running a project operation.

The small PowerShell vocabulary

From a normal PowerShell terminal, use `Get-Location` to print the current folder and `Get-ChildItem` to list it. Use `Set-Location 'C:\BookLab\sensor-monitor'` to enter your actual extraction root. Quotes keep a path with spaces together. A leading dot in `.\scripts\replay.cmd` means the current folder; `..` means its parent. Do not copy a prompt decoration such as `PS C:\...>` as part of a command.

Use `Get-Command git` to see whether the shell can resolve Git. Replace the tool name for another executable. A tool may be installed in a documented portable directory without being on the current `PATH`, the list of locations searched for commands. Follow the project's setup instructions or use its explicit environment override instead of scattering executable copies into unrelated folders.

After a native command, inspect `$LASTEXITCODE` promptly. The value belongs to the most recently executed native program and can be replaced by a later command. A result of zero ordinarily indicates successful completion for these project wrappers; the application's documented meaning is authoritative. An intentional `FAULT` state in synthetic replay can still be a successful program execution. A failed assertion and a fixture parse error are different outcomes.

```
# Context: PowerShell in the applicable project checkpoint root.
.\scripts\replay.cmd
$LASTEXITCODE
```

Keep the exact command, source checkpoint, and output together. If you redirect output to a file, use a deliberate path inside your evidence folder and do not overwrite the only copy of an earlier result. Read error output as well as ordinary output. A command that never started is not a failed policy assertion.

Windows ARM setup boundary

The recorded native compiler is the ARM64 LLVM-MinGW package identified in `dependencies.json`. Keep the package's directory structure. The `.cmd` wrappers manage the selected compiler's runtime directory for their own process, avoiding a permanent user/system `PATH` change. The primary path does not require a PowerShell execution-policy change. Optional `.ps1` wrappers apply only where the existing policy permits them.

If automatic discovery does not match your installation, set the documented overrides in the current PowerShell session using your actual executable paths:

```
# Context: PowerShell; examples must be replaced with actual installed paths.
$env:CXX = 'C:\Tools\llvm-mingw\bin\clang++.exe'
$env:ARDUINO_CLI = 'C:\Tools\arduino\arduino-cli.exe'
.\scripts\replay.cmd
.\scripts\build-firmware.cmd
```

A native host compiler and ESP32 cross-compiler are distinct. The first builds the program running on Windows; the second builds firmware for the selected board. Read the [actual setup](#) and [dependency manifest](#) for pinned archives, hashes, installation locations, and full target options.

macOS host appendix path

The reference project provides a Python wrapper for a host C++17 compiler. The authoring host path was exercised with Apple Clang 21.0.0 and Python 3.13.12 as recorded in project evidence. It is separate from the Windows learner demonstration and does not certify a different firmware or physical path. From the final project root:

```
# Context: macOS shell in CP09/final sensor-monitor, with documented tools installed.
python3 scripts/native.py test
python3 scripts/native.py replay
python3 scripts/native.py commands
```

Earlier checkpoints may not contain these later helpers; use their README. Do not translate a Windows path by changing slashes and assume every compiler option or dependency remains available. Record the actual version and result on your platform.

Linux guidance, not a tested certification

The host code is designed for a C++17 compiler, and the Python wrapper supports an explicit compiler selection. Linux users can inspect the wrapper and run it with their installed compiler, such as the documented `--cxx` option for g++. This edition's Windows and macOS results do not constitute a Linux execution receipt. Complete a new setup record and relevant checks before describing that combination as tested.

For any platform, keep the source unchanged while investigating a loader or path failure. Once the executable reaches a reproducible program error, move to the software diagnosis workflow. This order prevents environment problems from causing unnecessary firmware edits.

B. Git task and recovery reference

Begin a recovery by naming the repository, current branch, working changes, staged selection, and recent history. Decide which content must remain recoverable. These commands are precise when their starting state is precise; they are not a menu of interchangeable ways to make an error disappear.

```
# Context: PowerShell in the repository you intend to inspect.
git status
git diff
git diff --cached
git log --oneline --decorate -5
```

Select or unselect content

`git add path` selects the current content of that path for the next commit. Inspect `git diff --cached` afterward. In a repository with an existing commit, `git restore --staged -- path` removes the selected difference from staging while preserving working content. It does not delete the file. Review the state again before committing. [git restore](#)

For the Chapter 5 harmless-output exercise, use the exact path `build-note.tmp`. Then add a narrow ignore entry. An ignore rule applies to matching untracked files; it does not erase tracked content or earlier history. If the artifact was already committed, stop and plan that different cleanup rather than assume an ignore entry solved it.

Discard one unwanted working edit

From a clean baseline, after making one explicitly unwanted unstaged README edit, inspect `git diff -- README.md`. If you intend to lose that edit, `git restore -- README.md` restores the working file from the index by default. The index equals the baseline in this prepared example. If useful staged or working differences exist, inspect them before using this recipe.

Avoid a whole-directory restore when the decision concerns one path. Copy any valuable text to a named safe location first or preserve it in an appropriate commit. Git does not promise to recover an editor buffer that was never saved or content that was never recorded.

Reverse an isolated committed change

Use `git show` to inspect the actual commit. In the prepared FAULT-hysteresis exercise, the tip is an isolated faulty comparison. Run its test to reproduce the failure, then `git revert --no-edit HEAD` after confirming that HEAD is the intended commit. Revert adds a new corrective commit; the original remains in history. Rerun the same relevant tests and cross-build. [git revert](#)

If later work overlaps the reversal, a conflict can occur. Inspect the conflict and intended behavior rather than choosing a side mechanically. A merge commit has additional parent

semantics and is outside this beginner recipe; do not apply an unexplained mainline option to a shared merge merely to make a command proceed.

Resolve or leave a merge in progress

Begin merges from a clean, understood state. When a text conflict occurs, read both contributions and write the intended final content. Remove markers, stage the resolved file, inspect the cached diff, and complete the merge commit. If you are not ready to resolve, `git merge --abort` is the documented route for abandoning the merge in progress. Its ability to reconstruct preexisting uncommitted changes is limited, which is why the clean start matters. [git merge](#)

Remote ahead

Preserve local work, run `git fetch origin`, and inspect `git log --oneline --graph --decorate --all -8`. If the local branch can simply advance, `git pull --ff-only` handles that case. Divergent histories require an integration decision. Do not treat force-push as a generic synchronization remedy. Verify the remote URL separately if the destination is wrong.

Advanced inspection: find a reference you moved

The reflog records local reference updates and may help locate a commit that is no longer named by the branch you expected. `git reflog --date=iso` is an inspection step. Read the relevant entry, inspect the candidate with `git show`, and create a new recovery branch at the verified commit if appropriate. Reflogs are local and subject to expiry; they are not a backup guarantee or a substitute for preserving work. [git reflog](#)

If you use a stash to set aside work, understand which files were included. The default does not necessarily include untracked or ignored artifacts. `git stash list` and `git stash show help` inspect recorded entries. Prefer applying and checking an entry before dropping it when preservation matters. The core chapters use explicit disposable folders and commits instead of relying on a stash for every interruption. [git stash](#)

The recovery is complete when the required behavior is restored, useful work remains accounted for, and the resulting history is understandable. A clean status alone cannot establish those three conditions.

C. Markdown and YAML reference

Markdown keeps technical notes readable as plain text while adding simple structure for a renderer. YAML is a data format used for small metadata blocks such as skill frontmatter. They serve different purposes: a Markdown heading organizes prose, while a YAML key supplies a value to a parser. Preview documents and validate the format consumed by the actual host.

A small README that teaches an action

This original example uses the constructs needed in the course:

```
# Synthetic monitor baseline

The host prints a fixed sample. No physical measurement is claimed.

## Run

From the extracted CP01 project root in PowerShell:

```powershell
.\scripts\replay.cmd
```

Read the [setup instructions](docs/setup.md) before running the build.

- Confirm the current folder.
- Read the actual output and exit status.
- Keep the synthetic label with the result.

## Evidence

Check	Result	Limit
Host replay	Record your actual run	No physical sensor observation
```

Use one main heading and meaningful subordinate headings. Blank lines separate paragraphs and blocks. Backticks mark inline code; fenced blocks preserve multi-line commands and their language label. A normal prose paragraph is not a reliable way to preserve code spacing. Core headings, links, lists, and fences are defined by CommonMark; tables and task lists are common extensions whose support must be checked in the chosen renderer. [CommonMark 0.31.2](#)

Write commands exactly as learners should enter them, with context outside or in a clearly marked comment. Do not include a terminal prompt as part of the command. If you show expected output, use a separate block and label it expected. If you show actual output, name the run that produced it.

Links and images

A relative link is resolved from the document containing it. In a root README, docs/setup.md points into the docs directory. In a document already inside docs, the equivalent sibling may be setup.md. Test the link from the rendered location instead of assuming a working filesystem path will be rewritten correctly by every export.

Use descriptive link text, such as “toolchain setup,” rather than “click here.” An image needs useful alternative text explaining the information it conveys. A screenshot caption should name the action and relevant result; decorative wording does not help a learner who cannot read tiny interface text. Keep selectable commands in the page even when a screenshot shows them.

YAML frontmatter for a skill

```
---
name: datasheet-to-constraints
description: Derive sourced constraints for an exact component from supplied primary documents.
---
```

The opening and closing delimiters mark the metadata block used by the skill format. Keys and values are separated by a colon and space. Use spaces consistently for indentation when nesting data. Quote a value when punctuation might be interpreted structurally, especially a colon followed by a space. YAML's full syntax is broader than this small subset; consult its specification when a parser reports an ambiguity. [YAML 1.2.2](#)

Do not put Markdown headings inside the metadata block or leave its closing delimiter out. Keep a skill's workflow in the Markdown body after frontmatter. A valid YAML document can still have the wrong keys or an overbroad trigger for the consuming tool; syntax and behavior require separate checks.

Recovery checks

If a code block swallows the rest of a document, inspect its closing fence. If a link fails, resolve it from the current document and check spelling/case. If a table renders as plain text, confirm the selected renderer supports that extension and that the separator row exists. If a skill is not discovered, check exact filename, location, metadata, and host behavior before rewriting its procedure.

Keep the final document useful without its author standing nearby. A reader should find the starting state, action, expected result, actual evidence, and remaining limitation. Formatting is successful when it makes those details easier to follow.

D. Reusable task-brief patterns

A reusable brief is a set of decision fields, not a spell that makes every answer correct. Keep the fields that affect the task and omit irrelevant ceremony. The downloadable [brief template](#) and examples below are original starting points; replace their context with the actual project state.

Inspection without edits

“From CP00, read the named source excerpt and technical note. Explain the input, output, and limits of what the code establishes. Make no edits or installation changes. Name the files actually used. Separate proposed commands from executed actions. Accept when each consequential claim is tied to code, a source, or an explicit unknown.”

Use this when understanding the current system is the task. It does not establish runtime access controls; inspect those separately. If the assistant proposes an implementation, keep it as a possible next task while completing the requested explanation.

One reporting change

“Starting from the clean CP01 baseline, change README's Report label: synthetic. to Report label: synthetic input. Preserve host source, firmware, fixed sample values, dependencies, and commands. Show the exact diff. Accept when the origin is clearer and only the intended README line changes. Report any check actually performed.”

This brief is intentionally narrow. A new dashboard or renamed output schema may be useful in another task but does not belong here. The small boundary lets the learner inspect the complete result before committing.

Diagnose before fixing

“The named fixture fails under the attached command. Expected behavior and actual diagnostic are included. Preserve the current failing input. Identify two plausible explanations and one small check that distinguishes them. Do not change policy merely to suppress the symptom. After evidence supports a cause, propose the smallest correction and rerun the original reproducer plus relevant regression checks.”

Attach the actual error, source version, and working directory. A hypothesis becomes useful when it predicts a distinguishing observation. Asking for several guesses without a test plan simply creates more possible edits.

Review a bounded diff

“Review the supplied diff against these requirements. Read nearby callers as needed. Report actionable findings with file/line, trigger, consequence, and source or test evidence. Distinguish observed behavior from prediction. Do not invent findings to fill a quota or rewrite unrelated code. If no material issue is supported, state that and identify only meaningful verification gaps.”

Use the resulting findings as review input. Check their evidence yourself before applying a correction. An assistant's agreement or disagreement is not a replacement for the requirement and actual program behavior.

Extract interface constraints

“Use the supplied exact board and component documents. Identify variant, revision, units, conditions, and relevant source locations. Return claim, value/unit, source, uncertainty, and design consequence. Separate chip limits from board connections and documented expectations from bench observations. Leave missing mappings explicit and do not infer physical wiring from a family name.”

This pattern asks for an artifact you can check row by row. It prevents a summary from hiding a missing unit or a mismatched board variant. The datasheet skill packages the same practiced procedure for repeated use.

Resume from repository context

“Use the current repository instructions, README, handoff, requirements, and relevant diff. Establish the actual state and identify the next bounded task. Preserve unrelated work. Run the documented applicable checks and report source identity, commands, results, and limitations. Do not treat an earlier handoff's pass statement as current evidence without its receipt or a relevant rerun.”

A good handoff makes this request possible without old chat history. If a referenced file is missing, report that gap and continue independent inspection. Do not invent a remembered decision to fill the space.

Write acceptance before the answer

For each pattern, include a normal case and a useful negative case. Specify the artifact and the evidence required to accept it. If you cannot decide whether a response meets the task, refine the requirement before requesting more implementation. The strongest prompt improvement is often a clearer expected behavior, not more emphatic language.

E. Project records that survive the conversation

A small set of records keeps different kinds of knowledge from collapsing into one long README. Requirements state desired behavior. Hardware references state sourced constraints. Decisions explain a chosen tradeoff. Test plans describe checks to run. Results record checks that actually happened. A handoff names the current state and next task. Link them where one depends on another.

Requirement example

An original requirement record can say: “R-EX: A missing synthetic sample selects FAULT and exposes no current temperature. Check by supplying a missing event and inspecting the output fields.” The requirement defines behavior; it is not a test result. A later result should identify which version implemented it and what the check observed.

The final reference project has its own [R01-R12 contract](#), including threshold equality, hysteresis, stale age, recovery, command grammar, shared source, and reproducibility. Do not copy a simplified introductory rule over that contract. Name the stage and requirement before changing behavior.

Hardware/source row

Use columns for claim, exact part/board, value and unit, conditions, source title/revision/location, confidence or unknown, and design consequence. If the board guide supplies a connection while the silicon datasheet supplies an electrical limit, keep those sources distinct. A row should not imply the same document established both facts unless it actually did.

An incomplete row is useful when it names what is missing: “Connector mapping unresolved; need schematic for the exact assembled revision.” A confident unsourced pin number is less useful because it hides the dependency. Physical observations belong in a separate result record with setup and method.

Decision record

Decision: Use synthetic input for the required workflow.
 Context: The course must be reproducible without buying or wiring hardware.
 Choice: Share policy/parser/output source between host and firmware builds.
 Consequence: Host tests and replay can check software behavior; physical acquisition, electrical operation, and sensor accuracy remain separate work.
 Revisit when: The optional physical adapter has a sourced plan and bench method.

This is an original explanation of the course design, not a bench result. A good decision record helps a future contributor understand why an apparently obvious alternative was not chosen. It need not become a long meeting transcript.

Test plan and result

Write expected values before execution. The final policy's clear boundary is 27.0 °C at the default threshold; the test should include equality because < and <= differ there. The expected value must come from the requirement, not from copying the implementation's current result.

Use this compact result format:

```
Check ID and requirement:
Date, environment, tools:
Source commit/hash and checkpoint:
Input origin and fixture:
Exact working directory and command:
Expected result:
Actual output/exit and evidence path:
Interpretation:
Remaining limitation:
```

A plan with empty actual fields is not failed documentation; it is simply a plan awaiting execution. A result must contain real observations. If a test cannot start because a dependency is missing, record that setup failure separately from the behavior you intended to test.

Handoff example

“Current source is the named reviewed commit. The README origin label is corrected. CP01 replay and cross-build results are saved at the listed evidence paths with their tool versions. No physical observation occurred. Next task: create a branch for the documentation improvement in Chapter 8 and preserve the current baseline. The unresolved item is a missing setup explanation for the compiler override.”

Replace every generic field with your actual state. A handoff should let the next person start without reading the entire history of attempts. Put current facts first and link diagnosis history if it remains useful. Do not mark a file accepted merely because a previous session said it was done.

Download the [record templates](#), then fill them as the project creates evidence. Empty templates are scaffolding; a useful knowledge base contains a small number of accurate, linked records whose claims can be checked.

F. Project instructions and focused skills

Use a prompt for a one-time task, a project instruction file for stable local guidance, a deterministic script for a repeatable computation, and a skill for a procedure with a clear trigger and inputs. These can support one another. A skill is not improved by containing every possible project rule, and a README is not automatically loaded into every agent session merely because it exists.

Project instructions

For the selected Codex path, `AGENTS.md` supplies project guidance according to the host's discovery and scope rules. Confirm the current official behavior and test it in a fresh session. Guidance is not access control and does not itself create a subagent. Runtime permissions remain a separate mechanism. [Codex project instructions](#)

A useful file names the project purpose, stage-appropriate commands, source-of-truth documents, change boundaries, and expected completion evidence. At CP03, the applicable baseline is replay/build, not the later test suite. At CP09, tests and parser fixtures are present. Linking to a requirement or setup record keeps the instruction short while preserving detail.

The complete reference [AGENTS.md](#) belongs to its current source stage. Copying it into a much earlier checkpoint without adapting the commands can create false instructions. The learner [instruction template](#) explicitly requires that adaptation.

Skill structure and current host behavior

A skill directory contains `SKILL.md`, with metadata naming the skill and describing when it applies, followed by the procedure. Focused references or deterministic helpers can live beside it. The shared Agent Skills specification defines the format; each host determines discovery, invocation, and execution behavior. A valid directory alone does not prove activation. [Agent Skills specification](#)

The project uses `.agents/skills/<skill-name>/SKILL.md` for its selected repository-local path. Check the current host documentation and observe actual behavior in a fresh task. A description should help the runtime select the relevant procedure while excluding unrelated work. Explicit invocation is useful when testing the procedure itself; a separate test asks whether selection happens appropriately.

Datasheet-to-constraints procedure

The complete editable package is in [learner resources](#), including its supporting extraction rules. Its job is to turn supplied primary documents for an exact component or board into a traceable constraints table. It requires identity and source applicability, separates chip limits from board connections, and preserves missing facts.

Test it with a positive case supplying an exact source excerpt and requesting interface constraints. Check whether each value has a unit, source location, and consequence. Then give an unrelated README punctuation task; the procedure should not inappropriately drive that task. Finally omit a necessary board mapping and check that the result reports the gap instead of inventing a pin.

These are three different tests: useful output for relevant input, appropriate trigger boundaries, and honest incomplete-input behavior. A single successful invocation does not establish all three. Save actual request, activation evidence where available, output, and review.

Firmware-review procedure

The complete editable [firmware-review skill](#) reads a bounded diff, nearby callers, and requirements. It checks actionable correctness issues such as timing arithmetic, invalid/stale/recovery paths, command bounds, conversions, and whether tests distinguish the required behavior. It reports findings with a trigger, consequence, file location, and evidence.

The positive test uses a meaningful seeded defect, such as missing clear equality. The negative test uses an unrelated documentation change. The incomplete-input test omits the governing requirement or executable environment and checks that the review states the limitation. A review that invents a problem to fill a finding quota fails the procedure just as a review that ignores a real boundary can fail it.

If no actionable issue is supported, a concise statement of that result and material verification gaps is appropriate. Do not let the skill silently rewrite the diff when the task was review. Separate observed behavior from predicted consequences and distinguish host evidence from physical checks.

Maintain the package

When a tool update changes discovery or command behavior, rerun the relevant behavior cases. Keep references short and current. If two instruction files conflict, identify their scope and the actual rule selected before changing several files at once. If a helper can compute something reliably, use it for that computation and let the skill explain the decision around it.

The success criterion is observed useful behavior: the right procedure, supported output, appropriate restraint, and a reproducible record. A plausible filename or attractive frontmatter is only the beginning.

G. Versions and checkpoint map

The project reference version is 1.0.0. A content tag, installed tool version, and accepted release status are separate facts. Use the final release manifest for distributed file hashes and the verification report for actual platform results. This table records the edition's source and pilot configuration; it is not a promise that every future product update behaves identically.

| Component | Recorded value | Interpretation |
|------------------------------------|---|---|
| Primary learner OS | Windows 11 ARM64 | Standard-user demonstration path |
| Editor | VS Code 1.137.0 ARM64 | Recorded installed editor |
| Codex extension | 26.908.40401 | Installed interface; actual task evidence is separate |
| Bounded CLI pilot | Codex 0.154.0-alpha.6.2 | CLI correction evidence does not imply an IDE capture |
| Arduino CLI | 1.5.1 | Portable Windows x86_64 binary used in ARM guest |
| ESP32 platform | esp32:esp32@3.3.11 | Cross-compiler and board platform dependency |
| Native Windows compiler | LLVM-MinGW 20260908, Clang 23.1.1, aarch64 UCRT | Host replay/test compiler and runtime distribution |
| C++ mode | C++17 | Shared host policy/parser/output compilation |
| Separate macOS host path | Apple Clang 21.0.0; Python 3.13.12 | Distinct software evidence path |
| Required physical sensor libraries | None | Required firmware uses synthetic acquisition |

The [dependency manifest](#) contains archive URLs and hashes, architecture, versions, and the complete FQBN. Do not abbreviate the target to a family name when reproducing a build. The selected options include CDC on boot, enabled QSPI PSRAM, 4 MB flash, 240 MHz CPU, QIO flash mode, and TinyUF2 no-OTA partition scheme. Those options belong to the documented example target, not every ESP32-S3 board.

Use the right stage

| Stage | Chapters / course | Contents and applicable check |
|-------|------------------------------------|--|
| CP00 | CH01-04 / M01 and setup inspection | Original excerpts/log; annotate; no build required |

| Stage | Chapters / course | Contents and applicable check |
|-------|-------------------|--|
| CP01 | CH05-07 / M02-03 | Fixed synthetic host sample and minimal firmware; replay/build |
| CP02 | CH08-09 / M04 | Project/hardware records; content/link checks plus baseline |
| CP03 | CH10 / M05 | Agent instructions; fresh-session replay/build behavior |
| CP04 | CH11-12 / M05-06 | Constraints skill/interface plan; source and activation tests |
| CP05 | CH13 / M06 | Normal/fault shared policy and synthetic adapter; replay/build |
| CP06 | CH14 / M06 | Reproducer and fault exercises; diagnosis/replay/build |
| CP07 | CH15 / M07 | Basic normal/fault assertions; test/replay/build |
| CP08 | CH16 / M07 | Review skill and handoff; fresh-session behavior |
| CP09 | CH17-18 / M08 | Threshold/hysteresis/commands and complete verification |

Each ZIP has a sensor-monitor top folder and a checkpoint note. Extract to a new exercise folder. A ZIP contains files, not a Git graph. The separate sensor-monitor-history.bundle provides generic manufactured history, content tags, conflict branches, and isolated fault tags. Preserve the release identifier with the bundle you use.

The current project README defines the exact available commands. Early CP01 replay prints a fixed synthetic JSON line; a fixture file present there is reserved for later use and does not mean the early program parses it. CP07 introduces assertions. CP09 introduces configurable threshold, hysteresis, and command behavior. A later source file appearing in an early exercise is a checkpoint mismatch to investigate.

Fault and recovery map

FAULT-fixture belongs to CP06 and contains a malformed numeric token. FAULT-stale derives from the basic test stage and disables expiration. FAULT-hysteresis and FAULT-command derive from the final project and alter clear equality and complete-token validation respectively. Their exact start and recovery instructions live in the [checkpoint guide](#).

Chapter 7 uses the prepared later FAULT-hysteresis only to teach reverting an isolated commit with supplied tests. That does not advance the learner's main project to capstone completion. Keep the recovery clone separate and resume the intended earlier checkpoint afterward.

Recheck unstable account eligibility, download links, extension discovery, and installation screens when producing a new edition. Preserve old results with their versions rather than

relabel them as fresh evidence. A maintained workflow has a dated supported path and a small set of regression checks, not an unqualified promise of permanent compatibility.

H. Troubleshooting by evidence boundary

Begin with the exact symptom, command, current folder, checkpoint, and most recent relevant change. Preserve a small reproducer and the original diagnostic. Then choose a test that distinguishes explanations. Randomly changing source, reinstalling tools, and increasing permissions at once makes the result harder to interpret.

| Symptom | First useful check | Targeted recovery |
|---|--|--|
| A relative file cannot be found | Get-Location, expected root contents | Enter the correct extraction root and rerun the affected command |
| Tool name is not recognized | Get-Command, documented install location | Use supported installation or current-session override; reopen terminal when needed |
| A .ps1 file is blocked | Existing execution policy and checkpoint README | Use the primary .cmd wrapper; do not weaken policy to follow the course |
| Native executable exits before output | Actual exit code and runtime location | Use the documented wrapper/compiler distribution; distinguish loader failure from a test failure |
| Firmware target is unavailable | Core list and manifest FQBN | Install the pinned documented platform and select exact options |
| Host check passes but cross-build fails | Compiler diagnostic and target source list | Investigate target compatibility separately; host success does not establish cross-build success |
| Replay reports fixture line error | Original row, grammar, line number | Correct malformed fixture input; do not disguise it as a sensor fault |
| Replay enters FAULT | Event, validity, freshness, recovery requirement | Decide whether FAULT is expected behavior before changing code |
| AI explains a missing file | Actual action record and folder scope | Correct context/path and repeat bounded inspection |
| Skill seems ignored | Exact path/name, metadata, host discovery | Test explicit invocation, then trigger behavior; simplify conflicting guidance |
| Git push fails | Remote URL, identity, fetched history | Separate auth, destination, and divergence; preserve work before integration |
| A new checkout cannot build | Tracked files and documented dependencies | Add missing source or setup instruction, then repeat the affected reproduction |

When an AI answer is the failing artifact

Isolate one consequential claim. Ask whether it is code-supported, documented, tested, inferred, or unresolved. Provide the exact relevant passage or source and request a corrected artifact. Do not make an answer true by silently changing the requirement to match it. If an alternative design is worth considering, record it as a proposed decision with tradeoffs.

If an agent makes unrelated changes, stop further work and inspect the actual diff. Preserve useful content and return to a known state using the appropriate Git operation. Narrow the next brief around a concrete artifact and acceptance criteria. The correction should remove the unrelated effect while retaining the requested improvement.

When a test fails

First confirm the test executed. A compiler error, loader failure, failed assertion, and valid FAULT-state replay are different events. Read the project [format contract](#): successful replay may intentionally produce FAULT, while malformed fixture syntax exits with a diagnostic. Do not equate every alarming word with the same failure.

Keep expected values independent of the current implementation. If you change both the program and its expected output to make a test pass, explain which requirement justified the new expectation. A seeded fault should fail before the fix and the original source should still pass afterward. That demonstrates a meaningful distinguishing check.

Optional serial and physical work

The required firmware uses synthetic acquisition and the core course does not require upload. If you choose the optional physical extension, identify the actual board variant, cable capabilities, host port mapping, documented boot mode, and adapter configuration. Check the manufacturer's current instructions. A charge-only cable, missing device mapping, or wrong port can prevent communication without any policy defect.

Do not describe a cross-build as an upload receipt. Record the actual board, source artifact, upload command, tool result, and observed serial behavior if performed. Keep physical connections and measurements separate from host simulation. A serial line received from a board can still contain deliberately synthetic values; origin depends on the firmware's acquisition path, not simply the cable.

Know when to pause one path

Some dependencies require a human login, an unavailable account allowance, a source document, or optional physical observation. Name the exact missing item and continue work that does not depend on it. A pending check is preferable to a fabricated pass. The goal is to finish the engineering task with traceable evidence, not to keep every tool busy regardless of what is known.

I. Sources, rights, and corrections

A technical reference is useful when a reader can find the claim, identify the applicable source, and understand the limit of what it establishes. Prefer primary documentation for product interfaces, specifications, and commands. Keep a document title, owner, revision/date when available, section/page, access date, and a short paraphrase beside a consequential fact.

A source record

```
Claim ID and affected chapter/lesson:
Exact statement being supported:
Source title and owner:
URL, revision/date, section/page:
Access date:
Short paraphrase with units/conditions:
Status: documented / inferred / tested / unresolved
Remaining applicability question:
```

The same URL can support one claim and leave another unresolved. A board guide may describe a connector while a chip datasheet describes electrical limits. A test report establishes a particular observed run. Do not collapse these into one broad label such as verified hardware.

The book links primary sources near technical claims. The [project hardware reference](#), [dependency manifest](#), and [requirements](#) keep exact source decisions close to the code that depends on them. When a source changes, inspect the affected claim and its tests before updating the project.

Asset and code provenance

Original diagrams explain relationships; they are not screenshots of application actions. Screenshots must record genuine learner surfaces, preserve source images, and carry captions derived from their actual pixels. Constructed examples and seeded faults are labeled as such. If a live agent behaves differently from a saved instructional response, keep the actual response and adapt the explanation rather than changing the displayed answer.

Record each asset's creator/source, purpose, applicable license or permission, source path/hash, derivative path, caption/alternative text, and review status. Preserve third-party notices with dependencies. Public distribution decisions belong to the release's rights statement; an author's private teaching bundle does not automatically assign a license to every third-party item it references.

For narration, preserve the final spoken text and provider/voice provenance with actual usage records. Captions and transcripts must match the final audio, including editorial changes. A script can be ready for recording while the narrated lesson remains incomplete. Keep those statuses separate in the release evidence.

Report an erratum

An effective correction report identifies edition, chapter/lesson, location, current text or behavior, expected correction, supporting source or reproducer, and impact. Remove credentials and unrelated private material from the report. Do not send a full project archive when a small permitted excerpt explains the issue.

Example: “In the named edition, the recovery example describes the 28.0 entry boundary, but the supplied fault tag changes clearing at 27.0. The actual diff and test identify the clear-equality case. Update the explanation and affected narration; the source fix itself is unchanged.” This separates the content error from the program behavior.

Maintain the edition

Keep a change log with version/date, affected artifacts, reason, and validation repeated. A corrected prose label may need link/layout review; a changed source requirement may need tests, replay, cross-build, screenshots, and narration updates. Repeat checks because their inputs changed or a concern remains, not simply because an edition number changed.

The final verification report should distinguish content completeness, software checks, optional physical results, privacy/rights review, accessibility/layout, media synchronization, reference-LMS behavior, and human learner feedback. Agent walkthroughs are useful technical evidence but should not be described as human learner trials. A private release can be complete within its stated scope while later public publication and destination-specific administration remain separate work.

Glossary

Use the chapter references to return to the worked example. Definitions describe the course's usage rather than every meaning a term can have.

| Term | Practical meaning |
|----------------------|---|
| Acceptance criterion | An observable condition used to judge whether a requested artifact or behavior is complete. CH03. |
| Acquisition | Obtaining a sample from an input source. The required project uses synthetic acquisition; a physical adapter is separate. CH13. |
| Agent | A workflow in which model output can lead to permitted tool actions such as reading, executing, or editing. CH01. |
| AGENTS.md | Project guidance recognized according to a host's discovery and scope rules; not access control or a subagent definition. CH10. |
| API | An interface through which software calls another component or service. The selected course sign-in path does not require a separately billed API client. CH04. |
| Application | The interface and surrounding features through which a model is used. CH01. |
| Artifact | A concrete output such as source, a brief, a test record, or a diagram. |
| Assertion | A check comparing actual program behavior with a required condition; failure produces test evidence. CH15. |
| Baseline | A named, understood starting state used for comparison and recovery. |
| Bench evidence | An observation from a real physical setup using a stated method. It is distinct from simulation. |
| Boundary case | An input at or around a condition where behavior changes, such as exact threshold equality. CH15. |
| Branch | A named line of Git history whose reference advances as commits are made. CH07. |
| Build | The process that produces executable artifacts from source and dependencies using a selected configuration. |
| Caller | Code that invokes a function and supplies its inputs. CH01. |
| Caption | Synchronized text for audio in video, or a figure's descriptive text depending on context. |
| Checkpoint | A supplied project stage with a defined starting state and applicable exercises. |
| Clone | A new local repository obtained from another repository or bundle. CH06. |

| Term | Practical meaning |
|-------------------|---|
| Code-supported | A claim directly supported by the supplied source within its stated assumptions. CH01. |
| Commit | A recorded selected snapshot with metadata and relationships to earlier Git history. CH05. |
| Compiler | A tool that translates and checks source under language and target rules; it does not measure hardware. |
| Constraint | A condition a valid solution must preserve, such as a unit, interface, or excluded change. CH03. |
| Context | Information available to a model for an interaction, including selected instructions, references, and tool results. CH02. |
| Context window | A bounded capacity for material available in a model interaction. CH02. |
| Cross-compilation | Building code for a target different from the host executing the compiler. CH04/13. |
| Decision record | A short explanation of a choice, its context, tradeoffs, consequences, and revisit condition. CH09. |
| Dependency | An external tool, library, or component required for an operation. |
| Diff | A comparison identifying content changes between selected versions or Git states. CH05. |
| Documented | Supported by an applicable identified reference; not necessarily physically observed. |
| Embedding | A numerical representation used to work with relationships among inputs. CH02. |
| Evidence | Material that supports a bounded claim: source, document, executed result, or physical observation. |
| Exit status | A program's completion code, interpreted according to that program's contract. |
| FAULT | A project state indicating unavailable, invalid, stale, or recovering input according to its requirements. |
| Fetch | Retrieve remote objects and update local remote-reference knowledge without integrating into the current branch. CH06. |
| Firmware | Software intended to run on an embedded target. |
| Fixture | Controlled input used to exercise a program. Supplied course fixtures are synthetic. |
| FQBN | Arduino's fully qualified board name/configuration identifier used to select a build target and options. |

| Term | Practical meaning |
|-----------------------|--|
| Frontmatter | A delimited metadata block at the beginning of a file, such as skill YAML. CH11. |
| Grounded claim | A claim traceable to relevant supplied or retrieved evidence. CH02. |
| Hallucination | Here, a generated factual claim that is unsupported or false, such as an invented API or source section. CH02. |
| Handoff | A concise current-state record with evidence, open questions, and next action. CH16. |
| Host | The computer running development tools or a native test program. |
| Hunk | A grouped section of a diff with surrounding context. CH07. |
| Hysteresis | Different transition boundaries depending on current state, used here to retain state inside a temperature band. CH17. |
| Illustrative | Constructed to explain an idea; not presented as an observed execution result. |
| Index | Git's staging area: selected content for the next commit. CH05. |
| Inference | Using a trained model to process new input and produce output. CH02. |
| JSON Lines | A text format with one JSON object per line, used for project output. CH13. |
| Markdown | Plain text with conventions for headings, links, code, and other rendered structure. CH08. |
| Merge conflict | Overlapping contributions requiring a deliberate content resolution. CH07. |
| Model | The learned system that processes input and generates output. CH01. |
| Mutation/seeded fault | A deliberate defect used to show whether a check detects meaningful wrong behavior. CH15. |
| Native executable | A program built to run on the current host environment. |
| Nonfinite | A floating-point value such as NaN or infinity, rejected as a valid reading by the final policy. |
| Permission | A runtime or operating-system rule governing what actions are allowed. Guidance alone is different. |
| Policy | Logic translating inputs and state into defined behavior, separated here from acquisition. |
| Post-training | Further model training intended to shape behavior after an initial training stage. CH02. |
| Primary source | The originating specification, official documentation, research paper, or direct observation supporting a claim. |

| Term | Practical meaning |
|------------------|---|
| Pull | Fetch followed by integration according to options and configuration. CH06. |
| Pull request | A collaboration surface proposing integration between branches, including a diff and review discussion. CH06/07. |
| Push | Transfer local commits and request a remote reference update. CH06. |
| Regression check | A check that required behavior remains correct after a change. |
| Remote | A named Git repository location stored in local configuration. CH06. |
| Replay | Executing software with a controlled sequence of supplied events and injected time. CH13. |
| Reproducer | The smallest useful input and sequence that demonstrates a reported behavior. CH14. |
| Restore | A Git operation changing selected working or staged content from another state, according to its options. CH07. |
| Retrieval | Bringing additional material into context for a task; interpretation still needs review. CH02. |
| Revert | A new Git commit reversing a selected earlier change. CH07. |
| Rollover | A finite-width counter wrapping from its largest value to zero; timing arithmetic must state its assumptions. CH15. |
| Sandbox | A runtime boundary restricting actions or access; exact behavior depends on the host configuration. |
| SCORM | A course packaging/tracking standard used for the companion LMS export. It does not certify engineering competence. |
| Skill | A focused reusable procedure with a trigger, inputs, instructions, and optionally references/helpers. CH11. |
| Source of truth | The designated applicable record governing a particular fact or requirement. CH09. |
| Staging | Selecting current content for the next Git commit. CH05. |
| Stale | Too old to be treated as current under a stated freshness requirement. |
| Synthetic | Deliberately generated input used for software exercises, not a physical measurement. |
| Target | The embedded platform selected for firmware compilation and optional deployment. |
| Task brief | A bounded request naming goal, context, constraints, output, and acceptance criteria. CH03. |
| Terminal | A text interface displaying shell commands and program output. CH04. |

| Term | Practical meaning |
|-------------------|--|
| Token | A unit used to represent model input/output, not reliably one word. CH02. |
| Training | Adjusting model parameters through data and an optimization process. CH02. |
| Transcript | Text matching the final spoken narration; an earlier script is not automatically the transcript. |
| Unresolved | Not established by the current evidence; should have a concrete next question or check. |
| Upstream | The tracked branch relationship used for synchronization/status context in Git. CH06. |
| Working directory | The folder from which a command resolves relative paths. CH04. |
| Working tree | The project files being edited in a Git checkout. CH05. |
| YAML | A data-serialization format used for the small skill metadata block. CH11. |