GitHub Copilot App for developers

Isolate work, check results, and hand off the next step

Session M2-05 | Module 2 | Intermediate

Agenda

Segment	Time
App mental model and engineering surfaces	10 min
Chats, sessions, modes, and isolation	15 min
Tests, diffs, and handoffs	12 min
Issue-to-PR outer loop and critique	13 min
Prepared demo and lab handoff	10 min

The app has two different work surfaces

Surface	Good use	Boundary
Chat	Questions, comparison, early exploration	No dedicated branch or working tree
Project session	Code changes, commands, tests, PR work	Runs in a selected project workspace

Decision: Start in chat to understand the task. Start a project session when you need file changes or results tied to a branch.

A useful prompt names the files and checks

Weak:

```
Fix the readiness tool.
```

Better:

```
Read issue-brief.md and the failing tests. Explain the current behavior, propose
the smallest change, then run the focused test command. Do not edit unrelated files.
```

The second prompt names the scope and the checks the agent must return.

Choose autonomy from the cost of a wrong move

Mode	Use it when	Human checkpoint
Interactive	Scope is unclear or decisions need discussion	Frequent steering
Plan	The task spans files or has review risk	Approve the plan before edits
Autopilot	Scope and checks are clear, and rollback is cheap	Review the diff and check results

Mode is a risk control. It is not a ranking of model quality.

Isolation has two layers

```
Project
├── local repository session
├── working-tree session A
└── working-tree session B
```

A working tree separates branches and files for concurrent sessions.

Local sandboxing can also restrict filesystem, network, and credential access.
Working-tree isolation does not provide that security boundary by itself.

Check the workspace before work starts

Ask the session to report:

```
pwd
git status --short --branch
git worktree list
```

Then check:

- the expected project path;
- the session branch;
- no unrelated changes;
- the allowed test command.

If the boundary is wrong, stop before editing.

Attach context for the next decision

Use `@` references or `/attach-files` for:

- the issue brief;
- the failing test or error output;
- the implementation file;
- the session handoff.

Do not attach the whole repository by habit. Unrelated context can obscure the files the task needs.

Make the handoff usable in a fresh session

Write a short handoff that includes:

```
Goal
Accepted scope
Files changed
Checks run and exact result
Open risks
Next decision
```

Start a fresh session and attach the handoff plus the relevant files. If the new session cannot continue, the handoff is incomplete.

Check behavior before editing

```
Observe failing behavior
  → form a bounded plan
  → change the smallest surface
  → run focused checks
  → inspect the diff
```

Check the result with a command, diff, test, or reproducible manual check.

Check only what supports the decision

Claim	Check	Result
Failed checks block readiness	focused unit test	pass
Unresolved blocking review blocks readiness	snapshot fixture + test	pass
Neutral checks do not hide failures	full test output	pass

Each claim should point to something another engineer can inspect.

Critique the plan before implementation

Use the required command while the plan can still change:

```
/rubber-duck Critique the plan and tests against issue-brief.md. Look for a false
READY verdict or missing reason-code checks. Do not edit files.
```

Check each suggestion. Revise the plan or explain why the suggestion was rejected.
Record the decision in `engineering-review.md`.

Review the patch after implementation

```
/review Review the uncommitted repair against issue-brief.md.
Focus on correctness and regression coverage. Cite files and lines. Do not edit.
```

Triage findings against actual code or tests. Fix supported defects and rerun tests and review. Keep the patch uncommitted until this is complete.

Plan critique and code review answer different questions.

Why rubber duck is a second opinion

It is a built-in, read-only critic in the CLI and app.
Copilot selects a contrasting model automatically when one is available.

Record the model only if the client shows it. Check the feedback yourself.
An extra reasoning pass adds usage; it does not approve the change.

Session M2-01 covers the CLI's persistent Proactive invocation opt-in.
This lab requires a manual invocation.

Fork a plan before writing code

```
/fork
/spar Challenge the readiness plan with a counterexample.
Use the same brief and tests. Reason only; do not implement.
```

Check the actual fork workspace and argument. Write only the checked `planning-alternative.md`, then inspect `/merge-to-parent`.

Accept only the note; check parent source is unchanged before plan approval.
Do not assume `/spar` has rubber duck's model policy or read-only guarantees.

The outer loop starts with issue intent

```
Issue acceptance criteria
  → branch and diff
  → pull-request summary
  → review threads
  → required checks
  → human readiness decision
```

A passing local test checks part of the change. Review threads and required checks still affect PR readiness.

Read the pull request as a linked record

Surface	Question
Issue	What outcome was requested?
Diff	What changed, including scope drift?
Review	Which concerns remain unresolved?
Checks	Which required jobs passed, failed, or did not run?

The app keeps these surfaces close. The learner still has to connect them.

Automate the report, not the decision

The lab defines a **manual, read-only** automation.

It may:

- read issue and PR metadata;
- inspect review and check status;
- return blockers, unknowns, and a suggested verdict;
- link each statement to its source.

It must not comment, label, push, approve, merge, or resolve threads.

Prepared demonstration

1. Open the starter as a project.
2. Use chat to explain the issue without editing.
3. Start a Plan-mode working-tree session.
4. Attach the brief and focused test.
5. Critique the plan, revise it, and approve it before implementation.
6. Run checks, inspect the diff, and run `/review`.
7. Show the synthetic PR snapshot and readiness report.

If the live app fails, show the saved session output at the same step.

Deliverables

You will submit:

- an implementation with passing tests;
- a session handoff;
- `engineering-review.md` with plan critique and code-review triage;
- the checked analysis note returned from the `/spar` fork;
- a PR-readiness report;
- a manual read-only automation specification.

Another engineer should be able to check the decision from the saved files and results.

Lab handoff

Check live app access before the lab. If access is unavailable, stop the lab. Saved session output may recover the trainer demonstration, but it does not replace learner app practice.

Use only the synthetic training files.

Continue with Session M7-05

The developer route uses `/orchestrate` for independent read-only reviews of:

- the readiness implementation;
- regression coverage;
- the supplied PR snapshot.

Redirect a child and reconcile conflicting readiness claims.
The product track's Sessions M7-01 to M7-04 and GitHub issue access are not required.