Context Workflows

Give each task the repository evidence it needs

Access and cost preflight

Use Enterprise Cloud as the governance baseline.

1. Verify current official GitHub documentation and customer administrator policy.
2. Confirm repository scope and approved data sources.
3. Use a non-sensitive, bounded question.
4. For metered work, define a customer-owned threshold, escalation route, and stop guard.

Context selection

Choose evidence a reviewer can explain:

- repository instructions;
- relevant source files and tests;
- architecture or API documents;
- approved task notes.

Do not include unapproved data merely to make a prompt more complete.

Context packet exercise

```
Task: add validation to an API endpoint

Packet:
- route file
- corresponding tests
- API contract
- repository conventions
```

Ask learners to explain why each source is needed and what is excluded.

Evaluate the result

- Does it cite or follow the selected repository evidence?
- Is the scope limited to the task?
- Are assumptions surfaced for human review?
- Do relevant tests pass?

No-access fallback

Assemble the same context packet manually. Use it to plan and review the change without a live surface, then compare the result with the stated acceptance criteria.

Agenda and time plan

Time	Topic	Trainer move
0:00	Why context fails	Diagnose a vague response
0:08	Context layers	Build a source hierarchy
0:18	Spaces	Curate and share evidence
0:30	Instructions and prompt files	Show repository artifacts
0:42	Memory and maintenance	Separate preview from policy
0:50	Before/after demo	Compare with a rubric
0:58	Lab handoff	Assign context packet

Transition: “Better output begins with better evidence, not a longer prompt.”

The context problem

Copilot cannot reliably apply information it cannot see. Extra context can create noise or expose data unnecessarily.

Common failure modes:

- the relevant contract is absent;
- stale documentation conflicts with source code;
- broad context hides the key constraint;
- local preferences are mistaken for team standards;
- restricted data is included without approval.

Context selection is an engineering and governance decision. Keep the packet focused.

A practical context hierarchy

```
Task intent and acceptance criteria
                ↓
Repository instructions and prompt workflow
                ↓
Curated Space: code, docs, issues, specifications
                ↓
Current file and explicit references
                ↓
Validated memory, when supported and approved
```

Specific task constraints override generic guidance. Clarify conflicts.

Copilot Spaces

A Space is a selected set of sources for a recurring topic or workflow.

Good candidates include:

- source repositories and key paths;
- architecture and API documentation;
- approved issues and specifications;
- team guidance for a domain;
- links whose ownership and freshness are known.

Do not use a Space to collect restricted information without approval.

Curate for a purpose

```
Space: Checkout API
├── API contract
├── checkout route and service
├── focused tests
├── error-handling conventions
└── approved migration notes
```

Exclude unrelated services, production data, secrets, abandoned proposals, and documents with no accountable owner.

Trainer cue: Ask learners what they would remove first if the answer became noisy.

Sharing Spaces safely

Before sharing:

1. Name the intended audience and task.
2. Confirm every source is approved for that audience.
3. Add purpose-specific instructions.
4. Assign an owner and review date.
5. Test with a known question and compare against source evidence.
6. Remove or replace stale sources.

Current access controls and sharing behavior must be verified before the demo.

Repository-level instructions

Create `.github/copilot-instructions.md` for durable project guidance:

```
# Repository guidance

- Use TypeScript strict mode.
- Validate request input at the route boundary.
- Keep business logic in `src/services/`.
- Run `npm test` before proposing completion.
- Do not add dependencies without reviewer approval.
```

Write rules that can be observed. Avoid vague goals such as “produce high-quality code.”

Reusable prompt files

Place team-reviewed workflows under `.github/prompts/`:

```
---
description: Review an API change against its contract
---

Read the selected route, service, tests, and API contract.
List mismatches before suggesting changes.
Do not edit files outside the selected feature.
End with the focused validation command.
```

Verify the current supported filename and frontmatter behavior before delivery.

VS Code instruction settings

The curriculum references:

```
{
  "github.copilot.chat.codeGeneration.useInstructionFiles": true
}
```

Treat settings as version-sensitive:

- verify current documentation and the learner’s IDE version;
- show where workspace settings are reviewed;
- do not claim an instruction loaded without observing the result;
- retain the instructions as a useful manual checklist.

Copilot Memory

Memory may let an agent retain and validate useful knowledge, depending on support and approval.

Use three questions:

1. **Source:** Where did this fact come from?
2. **Scope:** Does it apply to this user, repository, or task?
3. **Freshness:** How will it be corrected when the code changes?

Never store secrets, restricted data, or guesses. Verify current availability and controls.

Demo: without grounded context

Prompt:

```
Add validation to the checkout endpoint.
```

Expected discussion:

- Which endpoint?
- Which inputs and error schema?
- Where is validation normally performed?
- Which tests prove the behavior?
- Which data is safe to include?

Do not score the response by confidence or length.

Demo: with a context packet

```
Using the checkout API contract, route, service, focused tests, and repository
instructions in this Space, propose the smallest validation change.

First list acceptance criteria and assumptions. Do not add dependencies.
End with the focused test command and cite the evidence used.
```

Compare outputs for correctness, scope, assumptions, evidence, and testability.

Concrete code target

```
export function parseQuantity(value: unknown): number {
  if (!Number.isInteger(value) || Number(value) < 1) {
    throw new ValidationError("quantity must be a positive integer");
  }
  return Number(value);
}
```

Ask whether the API contract permits numeric strings. The correct next action may be clarification rather than code generation.

Maintenance is part of context

Trigger	Maintenance action
API contract changes	Update Space sources and prompts
Repository convention changes	Review instruction files
Ownership changes	Assign a new maintainer
Output contradicts source	Remove stale evidence and retest
Feature behavior changes	Recheck official documentation

Schedule reviews; do not wait for failures.

Lab handoff

Learners will:

1. curate a Space for the provided project;
2. add code, documentation, and specifications intentionally;
3. write repository instructions and three prompt files;
4. compare no-context, instructions-only, and Space-grounded responses;
5. document source ownership and a maintenance trigger.

Deliverable: A reviewable context setup with a before/after comparison.

Remember

1. Relevant context matters more than a large context packet.
2. Spaces curate evidence for a purpose and audience.
3. Instructions encode durable rules; prompt files encode reusable tasks.
4. Memory requires source, scope, freshness, and current policy checks.
5. Every context source needs an owner and maintenance path.

Questions and lab readiness

Which source would you remove from a context packet, and why?

- Which source is authoritative?
- What must stay out of the Space?
- Who will maintain the resulting artifacts?