Introduction to Copilot

Responsible AI-assisted development

The durable workflow

```
Define a bounded task
        ↓
Check access, data, and policy
        ↓
Inspect the proposed change
        ↓
Run tests and human review
        ↓
Accept, revise, or implement manually
```

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, participant access, and data classification.
3. Use a non-sensitive task with explicit acceptance criteria.
4. For metered work, set a customer-defined threshold, escalation route, and stop guard.

Good task descriptions

```
Add validation to the account-creation form.

Acceptance criteria:
- Name and email have clear validation errors.
- Existing tests pass and focused tests cover failures.
- No sensitive data is introduced.
- A human reviewer can reproduce the checks.
```

Clear scope gives reviewers something concrete to check.

Inspect, test, review

Ask:

- Does the change meet the acceptance criteria?
- Is the code understandable and within the approved scope?
- Did required tests and checks run?
- Does a reviewer understand data, dependency, and security implications?

Treat assistant output as a proposal. Approval still belongs to the reviewer.

No-access fallback

When access is absent or policy does not permit a live exercise:

1. Complete the starter task manually.
2. Use the same acceptance criteria and tests.
3. Note where an assistant might have helped.
4. Compare the result with the solution or trainer review.

Agenda and time plan

Time	Topic	Trainer move
0:00	Outcomes and mental model	Set expectations
0:05	Architecture and data flow	Draw the request path
0:15	Access, policy, and usage	Run the preflight
0:25	Setup and core controls	Walk through the IDE
0:35	Live coding demo	Narrate accept/reject decisions
0:48	Next Edit Suggestions	Show the follow-up loop
0:53	Strengths, limits, objections	Invite discussion
0:58	Lab handoff	Confirm readiness

Transition: “Start with a simple system model.”

What GitHub Copilot is

Copilot is an AI coding assistant available through development surfaces such as the IDE, GitHub, and the command line.

- It predicts or proposes code from the context available to the selected surface.
- It can explain, transform, test, and review code when asked.
- It does not know your intent unless the task and context make that intent clear.
- It does not replace compilation, tests, security checks, or accountable review.

Trainer cue: Ask learners to name one decision they would never delegate.

A useful architecture model

```
Developer intent + approved context
                ↓
        Copilot product surface
                ↓
      selected model and service
                ↓
           proposed output
                ↓
      local checks + human review
```

This is a conceptual model, not a promise about a fixed implementation.
Verify current product and data-flow details in official documentation.

LLM basics without the mythology

- A large language model estimates useful continuations from patterns in data.
- The prompt includes instructions plus context exposed by the active surface.
- The same request can produce different outputs as context or models change.
- Fluent output can still be incomplete, insecure, or incorrect.
- Feedback comes from accepting, rejecting, editing, testing, and refining.

Say: “Probability produces a proposal; engineering evidence produces confidence.”

Privacy, context, and telemetry

Before a demonstration, distinguish these questions:

Question	Evidence source
What context may be sent?	Current product documentation and surface behavior
What repositories or paths are allowed?	Administrator policy and content exclusions
What telemetry is retained?	Current plan, policy, and privacy documentation
May this task use real data?	Data owner and classification policy

Never paste secrets, credentials, personal data, or restricted source into a prompt.

Usage and billing: teach a method

Do not memorize prices, allowances, or model tables for delivery.

1. Open the current official billing and model documentation.
2. Identify the participant's plan and enabled features.
3. Determine whether the planned activity is metered.
4. Agree on a threshold, owner, alert, and stop condition.
5. Record what was checked and when.

Transition: "Once permission is clear, configure the working surface."

Supported environments

Availability changes, so demonstrate a verification workflow:

- Check the current supported IDE and platform documentation.
- Confirm the extension or integration comes from the approved publisher.
- Confirm the learner is signed into the intended account.
- Confirm the organization has granted the required seat and policy access.
- Use the language learners already know; coverage quality varies by task and context.

The lab uses VS Code, but the review habits transfer to other approved surfaces.

Installation and activation walkthrough

1. Open the Extensions view and locate the approved GitHub Copilot extension.
2. Install it and follow the sign-in flow.
3. Confirm the intended GitHub account and organization.
4. Open Copilot status and verify that suggestions are enabled.
5. Open the training repository and a non-sensitive utility file.
6. Type a small comment and pause for a suggestion.

Demo check: Have a screenshot or manual implementation ready if access fails.

Inline suggestion controls

Use the key bindings displayed by the learner’s current IDE:

- **Accept** only after reading the complete suggestion.
- **Reject** when scope, behavior, or style is wrong.
- **Cycle** when alternatives may better fit the task.
- **Accept partially** when only a bounded portion is useful.
- **Undo and revise the prompt** when the direction is wrong.

Read the proposal, then decide whether the evidence supports accepting it.

Live demo: bounded function

Start with intent and edge cases:

```
def normalize_username(value: str) -> str:
    """Return a lowercase username with surrounding whitespace removed."""
```

Ask Copilot to suggest the body, then inspect it against:

```
assert normalize_username("  Ada ") == "ada"
assert normalize_username("") == ""
```

Narrate every acceptance, rejection, edit, and test.

Demo narrative and recovery

Before typing: State the behavior and non-goals.

While suggesting: Ask, “What context appears to influence this output?”

Before accepting: Read the code aloud and predict each test result.

After accepting: Run the focused tests and inspect the diff.

If no suggestion appears: Implement the two-line body manually and compare the same review process. The learning objective is judgment, not a particular output.

Next Edit Suggestions

Next Edit Suggestions can propose likely follow-up edits after a change.

```
Change a function signature
          ↓
Inspect a proposed caller update
          ↓
Accept, reject, or refine
          ↓
Run tests and search for missed callers
```

Review each follow-up separately. A predicted edit can still miss a caller.

Where Copilot helps

- Repetitive, well-specified transformations
- Familiar language and framework patterns
- Test case brainstorming
- Explaining unfamiliar code as a starting point
- Drafting documentation and examples

Where judgment dominates

- Ambiguous product requirements
- Security, privacy, legal, and architecture decisions
- Novel domain rules and hidden dependencies
- Correctness claims without executable evidence

Common objections: a Q&A framework

Concern	Productive trainer response
“Will it replace developers?”	Focus on changed tasks, review, and accountability.
“Is the output always correct?”	No; demonstrate tests and rejection.
“What happens to our code?”	Use current official docs and customer policy.
“Does it work in every language?”	Verify support and test the actual workload.
“How much does it cost?”	Use live plan and billing evidence, never static claims.

Invite concerns; do not improvise policy answers.

Lab handoff

In the lab, learners will:

1. verify installation, sign-in, and policy access;
2. accept, reject, cycle, and partially accept suggestions;
3. experiment with Next Edit Suggestions;
4. implement small functions in multiple languages;
5. compare assisted and manual results with the same tests;
6. record where suggestions helped and where review caught issues.

Deliverable: A reviewed utility file plus an evidence-based reflection.

What to remember

1. Copilot proposes; developers remain accountable.
2. Context, instructions, and task boundaries shape suggestions.
3. Access, data, policy, and metering are preflight decisions.
4. Accept/reject controls and tests are core skills.
5. Current official documentation beats memorized product claims.

Questions and lab readiness

What would make a proposed change safe and reviewable in your repository?

- Which context is safe to use?
- Which check proves the demo function works?
- Who needs help confirming installation or access?