



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Welcome

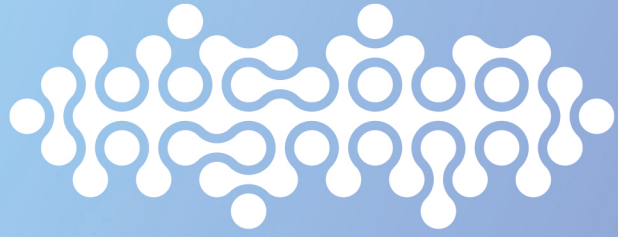
FROM GREENFIELD TO
BROWNFIELD

Today's Agenda

Session 1 (today):

0. **Workflow Refresher** — Git, PRs, hooks, AI tools
1. **Closing the Loop** — Stack Overflow Challenge Round 2 feedback
2. **The Brownfield Reality** — what changes when the code already exists
3. **Entering Locust** — our real-world codebase + live demos
4. **Project Context Files** — CLAUDE.md, copilot-instructions.md in action
5. **Getting Started** — clone, setup, first exploration

Session 2 (next): Measuring success + 60 min hands-on exercises



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Professional Workflow Refresher

0

The Feature Branch Model

Golden Rule: Never push directly to main.

Every change lives on its own branch. The `main` (or `master`) branch is always in a deployable state.



- Multiple developers work in parallel without stepping on each other
- Changes are reviewed before they reach main
- Broken code stays isolated on the feature branch

Covered in depth in [Software Engineering Lecture 3](#) — [full course](#)

The Workflow in 6 Steps

branch → commit → push → PR → review → merge

1. **Branch** — `git checkout -b feature/my-change`
2. **Commit** — Small, focused changes with clear messages (*hooks run here!*)
3. **Push** — `git push -u origin feature/my-change`
4. **Pull Request** — Open on GitHub, describe what and why
5. **Review** — Teammate reviews (*CI runs here!*), requests changes or approves
6. **Merge** — PR merged into main, branch deleted

Steps 2-3 repeat — commit and push as you go, not just once at the end.

We'll cover **hooks** and **CI** in detail on the next slides.

Branch Naming Conventions

Prefix	Use case	Example
feature/	New functionality	feature/team3-stats-coverage
fix/	Bug fixes	fix/percentile-overflow
refactor/	Code improvement	refactor/extract-logging-helpers

Rules:

- Lowercase, hyphens between words
- Descriptive — a teammate should understand the purpose from the name
- Short — nobody wants to type feature/add-comprehensive-test-coverage-for-the-stats-module

What Is a Pull Request?

A PR is a **communication artifact**, not just a code submission.

Every PR should explain:

- **What** changed — summary of the diff
- **Why** it changed — motivation, issue reference
- **How to test** — steps a reviewer can follow

AI can draft PR descriptions — but verify they describe what the code **does**, not what it **should** do.

Branch Protection

Our Locust repo has branch protection on `master`:

- **Requires 1 approving review** before merging
- **No direct pushes** — every change goes through a PR
- Admins can bypass (instructor only)

If `git push origin master` is rejected — **that is intentional**, not broken.

This simulates real open-source projects where maintainers review every contribution.

Pre-Commit Hooks

Automated quality gates that run before every `git commit`.

The Locust repo has this in `.pre-commit-config.yaml`:

```
repos:
- repo: https://github.com/astral-sh/ruff-pre-commit
  hooks:
    - id: ruff          # Linter
      args: [--fix]
    - id: ruff-format   # Formatter
```

If your code has PEP 8 violations → **commit rejected**. Fix the violations, then commit again.

This is the system **working correctly**, not a problem.

CI/CD in 30 Seconds

CI (Continuous Integration): Runs tests + linting automatically on every push/PR.

In a professional project: Nothing merges without CI passing.

In our teaching repo: CI is **disabled** (GitHub Actions quota).

This means: **You are the CI.**

```
# Run tests before every push
uv run pytest locust/test/test_stats.py -x -q

# Run linter (or rely on pre-commit hooks)
uv run ruff check . && uv run ruff format --check
```

Reading pytest Output

```
locust/test/test_stats.py::TestRequestStats::test_log ..... PASSED
locust/test/test_stats.py::TestRequestStats::test_extend .. PASSED
locust/test/test_stats.py::TestStatsError::test_parse ... FAILED
```

- **PASSED** — test verified the expected behavior
- **FAILED** — assertion did not hold. Read the output to see what was expected vs actual.
- **ERROR** — test could not run (import error, missing fixture, etc.)

Useful flags: `-x` stops on first failure, `-q` reduces noise, `-v` shows details

Your AI Tools Today

GitHub Copilot

- Chat panel + inline completions
- Multiple models (GPT-4o, Claude, Gemini)
- Reads `.github/copilot-instructions.md`
- You already know this tool

Claude Code *(new)*

- Terminal panel in VS Code (or standalone CLI)
- Claude models (up to 1M token context)
- Reads `CLAUDE.md` from project root
- Install: `npm install -g @anthropic-ai/claude-code`

Both can: read/write files, run commands, execute tests, make commits in **Agent mode**.

Interaction Modes

Mode	What it does	When to use
Ask	Answers questions, no file edits	Understanding code, exploring architecture
Agent / Edit	Reads, writes, runs commands	Writing tests, refactoring, fixing bugs
Plan	Designs an approach, no changes until you approve	Complex multi-step tasks, architecture decisions

For today's exercises: Agent mode is your primary mode.

The AI reads your `CLAUDE.md` (or `.github/copilot-instructions.md` for Copilot), understands the project, and can run tests to verify its own work.

Deep-Dive Resources

Git & Workflow:

[Feature Branch Development](#) — SE course lecture covering Git fundamentals, PRs, branch protection, CI/CD

AI Tools:

[Lecture 2 Slides](#) — Copilot vs Claude Code comparison, setup, permissions

If you have taken the SE course, this chapter was review. If not — bookmark these links.

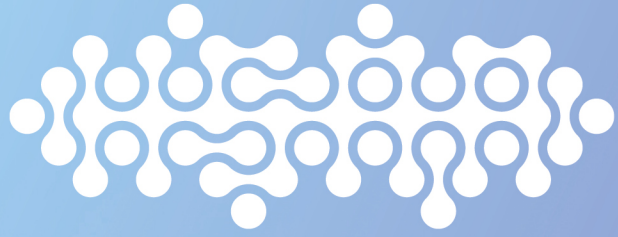
Quick Poll

Rate your comfort with this workflow (1-5):

1. "What is a branch?"
2. "I've heard of PRs but never made one"
3. "I can do the basic workflow but need reminders"
4. "Comfortable — I use this regularly"
5. "I could teach this"

Raise your hand for your number.

This tells us how much support you'll need during the exercises.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Closing the Loop

1

Feedback Quiz (5 min)

Open the Google Forms link: forms.gle/2ZZJsCvr3dgfAKsm9

Key questions:

- Which approach did you use? Strictly Approach 3 or blended?
- Did it improve? (1-5 scale)
- Iteration count: how many prompts? Compare to Round 1.
- AI Diary: overhead or useful?
- **Confidence for unfamiliar code?** ← Hold onto this answer.

What We Expect to See

Iteration counts go down. Approach 3 front-loads thinking. Each prompt is scoped and specific.

Confidence goes up. Not because the AI is better — because the *process* provides checkpoints.

Diary friction shifts. Writing the entry is quick now. The real friction: **commit discipline** — one commit per AI interaction, structured messages. That is not overhead — it is communication.

Blended approaches are fine. The point: *how* you interact with AI affects *what* you get.

A Small Addition to the Reference Repo

Open the reference repo: github.com/l3gj0n/so-challenge. Look at the root.

Two new files:

- **CLAUDE.md** — 45 lines of AI context
- **.github/copilot-instructions.md** — same content for Copilot

These also define the **diary entry format** — so you never have to spell out the structure in your commit messages. Just say "write a diary entry" and the AI references its own context file.

For a 250-line project, ~500 tokens of context. **Later today:** a CLAUDE.md for a 10,000-line project. Same concept. **Completely different scale.**

The Question That Launches This Lecture

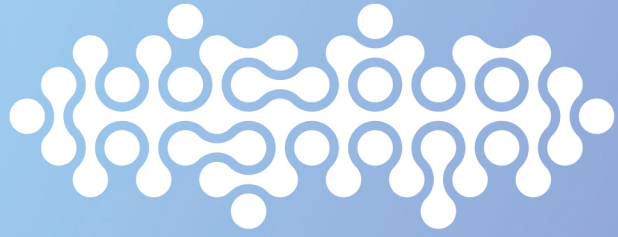
Look at the feedback results for:

"How confident are you about using AI tools in a codebase you did NOT write?"

If the pattern holds: **lower than your general confidence.**

That is an honest, accurate assessment. Everything so far was **greenfield** — you started from an empty directory, you made every decision.

But that is not what most software jobs look like.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The Brownfield Reality

2

How Developers Actually Spend Their Time

60-80% of professional software development effort goes to **maintaining and evolving existing systems**.

70% of development time goes to **program comprehension** — reading code, navigating call hierarchies, building mental models.

(Minelli, Mocci & Lanza 2015; LaToza, Venolia & DeLine 2006)

The primary developer activity is not writing — it is **reading**.

Greenfield vs Brownfield: 5 Key Differences

Greenfield (SO Challenge)

- You control the context
- Tests are aspirational
- You write more than you read
- "Done" = it works
- You set the conventions

Brownfield (Locust)

- Context is massive, AI needs help
- Tests are a safety contract
- 70% reading, 30% writing
- "Done" = checklist of 7 items
- Conventions are pre-existing

AI in Brownfield: More Useful AND More Dangerous

Where AI excels:

- Code explanation ("What does this function do?")
- Test generation ("Write tests for this module")
- Refactoring ("Split this 99-line method")

Where AI struggles:

- Architectural decisions (doesn't know *why*)
- Unwritten conventions (can't follow the implicit)
- Scope control (suggests bigger changes than needed)

The pattern: AI struggles are symptoms of *implicit knowledge*. Make it explicit → AI follows it.

Making Implicit Knowledge Explicit

Once a convention is written down — in a `CLAUDE.md` (or `.github/copilot-instructions.md`), a Definition of Done, an architecture doc — AI becomes significantly better at following it.

"Always use the existing `StatsEntry` class, never create a parallel stats structure."

This is exactly the kind of rule an AI can follow consistently, across hundreds of lines, without forgetting.

But "better" is not "perfect." More on this soon.

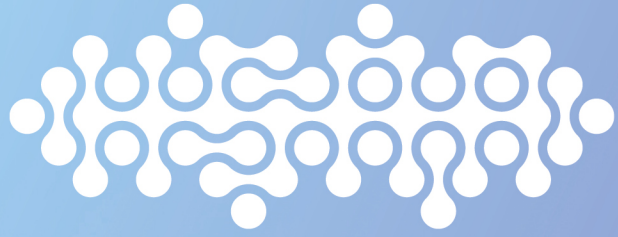
Discussion

Think about a project you have worked on (internship, thesis, side project, or even a group assignment):

What implicit conventions existed that were never written down?

- A naming pattern everyone followed but nobody documented?
- A module that "everyone knows" not to touch?
- A testing pattern that is used everywhere but not in a style guide?

2 minutes: discuss with your neighbor.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Entering the Locust Ecosystem

3

What Is Locust?

A Python load testing framework. ~25,000 GitHub stars, active development.

- Write user behavior scripts in plain Python
- Simulates thousands of concurrent users (gevent greenlets)
- Real-time web UI at `localhost:8089`
- Distributed mode across multiple machines

Why Locust for this course? Performance IS the product. Right-sized codebase (~10K lines). Uses pytest. Active and real.

Our Teaching Copy

github.com/hs-aalen-software-engineering/locust

- **Full import** of `locustio/locust` — complete commit history (6,000+ commits)
- **No connection** to upstream — your PRs stay here
- **Actions disabled** — you run tests locally
- **Branch protection** on `master` — 1 review required
- **CLAUDE.md** + **copilot-instructions.md** — AI context pre-configured

This simulates open-source contribution without risk.

Directory Structure

```
locust/
├── locust/                # Core Python package
│   ├── __init__.py       # Public API + gevent monkey-patching
│   ├── stats.py          # Request statistics (1,250 lines)
│   ├── runners.py        # Local/Master/Worker runners
│   ├── web.py            # Flask web UI + REST API
│   ├── user/             # User model + @task system
│   ├── contrib/          # Protocol clients (mqtt, postgres...)
│   └── test/             # Test suite (~700 tests)
├── examples/             # 40+ example locustfiles
├── CLAUDE.md              # AI context for Claude Code (615 lines!)
├── .github/copilot-instructions.md # AI context for Copilot
└── .pre-commit-config.yaml # ruff hooks
```

Key Modules

Module	Lines	What it does
<code>stats.py</code>	1,250	Response times, percentiles, aggregation
<code>runners.py</code>	1,500+	Spawning users, distributed mode
<code>web.py</code>	800+	Dashboard UI, REST API
<code>user/users.py</code>	300+	User/HttpUser base classes
<code>user/task.py</code>	500+	@task decorator, TaskSet
<code>clients.py</code>	550	HttpSession with stats integration

Tests live **inside** `locust/test/` — not a separate `tests/` directory.

Live Demo: Running a Load Test

```
# locustfile.py
from locust import HttpUser, task, between

class QuickStartUser(HttpUser):
    wait_time = between(1, 3)
    host = "https://httpbin.org"

    @task
    def hello_world(self):
        self.client.get("/get")
```

```
uv run locust -f locustfile.py
# Open http://localhost:8089
```

Only target `httpbin.org` or `localhost`. Testing servers you do not own may have legal consequences.

Let's try it live. If the web UI looks broken (raw text), the React frontend needs building — ask your AI tool how to fix it.

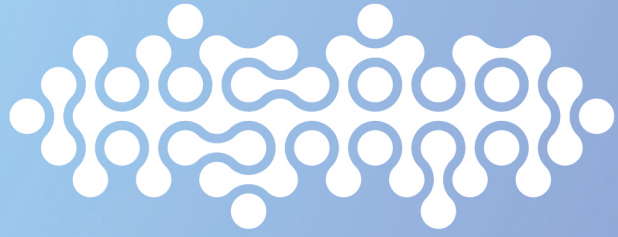
Live Demo: Running the Fast Test Subset

Not all tests are fast. Some take 2+ minutes. Use the curated subset:

```
uv run pytest locust/test/test_env.py locust/test/test_tags.py \  
locust/test/test_users.py locust/test/test_locust_class.py \  
locust/test/test_wait_time.py locust/test/test_util.py \  
locust/test/test_html_filename.py -x -q
```

125 tests, ~15 seconds, all clean. Your exercise packages have even faster commands (~8-22s).

Remember: **You are the CI.** Run tests before every push.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Project Context Files

4

The Context Problem

You just opened a **10,000-line codebase**. You want your AI to help.

But the AI knows **nothing** about this project. It does not know:

- That `stats.py` handles percentile calculation
- That tests use `unittest.TestCase` base classes, not pytest fixtures
- That tests live in `locust/test/`, not `tests/`
- That GitHub Actions are disabled

Without context, the AI gives **generic, often wrong** answers.

Why Project Context Files Matter

Modern AI models are smart enough to explore a codebase on their own. But context files give them a **head start**:

- Without: the AI reads files, infers patterns, *eventually* gets there
- With: the AI **immediately** knows the conventions, test patterns, and pitfalls

The difference is not right vs wrong — it is **fast and consistent** vs slow and ad-hoc. At 10K+ lines, that head start matters.

600 lines of structured context save minutes on every interaction.

Anatomy of the Locust CLAUDE.md

Open them now in the project root: `CLAUDE.md` + `.github/copilot-instructions.md`

Section	Purpose
Collaboration Principles	Behavioral instructions for the AI
Key Files to Understand First	Ordered onboarding guide (10 files)
Testing Details	Base classes, patterns, test commands
What NOT to Touch	Actions disabled, built artifacts, etc.

The SO challenge version: 45 lines. This one: **615 lines**. Same concept, different scale.

Real-World Adoption

This is not a teaching construct. Major open-source projects do this:

Project	Stars	CLAUDE.md
<u>React</u>	244K	Minimal — monorepo split note
<u>Deno</u>	106K	Detailed — architecture, debugging, testing
<u>Next.js</u>	138K	Symlink to AGENTS.md
<u>Zed</u>	78K	Coding style rules ("no abbreviations")

Project	Stars	copilot-instructions.md
<u>VS Code</u>	165K	Architecture, coding guidelines, testing
<u>.NET Runtime</u>	15K	Build system, conventions, contribution rules

Same principle, different tools. Our Locust repo has **both**.

What CLAUDE.md Does NOT Guarantee

A real example from preparing this course:

We asked the AI to generate a benchmark script. The project context file (`CLAUDE.md` / `copilot-instructions.md`) explicitly says "run ruff before committing." The result:

- Benchmark code **worked correctly**
- But had **PEP 8 violations** the linter would have caught
- Follow-up fix: AI fixed the code but **forgot the diary entry**

Project context files are influence, not enforcement.

The Three-Layer Model

Layer	What	How
Influence	CLAUDE.md / copilot-instructions	Probabilistic — AI <i>usually</i> follows
Enforcement	Pre-commit hooks, linters, CI	Deterministic — ruff passes or fails
Human	Your discipline, code review	Judgment — catches semantic issues

All three layers work together. In our repo, CI is disabled — so **hooks + you** are the safety net.

Even in the age of AI-assisted coding, automated checks remain essential.

Live Demo: Pre-Commit Hook in Action

Writing code with a deliberate PEP 8 violation...

```
git add benchmarks/run_benchmarks.py  
git commit -m "Add benchmark"  
# ruff rejects the commit!
```

Fix the violation → commit succeeds.

This is your safety net. The AI won't always follow the style guide. The hook catches it.

Tool Equivalents

Tool	Instruction File	Location
Claude Code	CLAUDE.md	Project root
GitHub Copilot	.github/copilot-instructions.md	.github/ directory
Cursor	.cursorrules	Project root

Same principle: Encode project context once, in a file the AI reads automatically.

The Locust repo has both `CLAUDE.md` and `copilot-instructions.md`. Whichever tool you use — you get the benefit.

Act II Tease

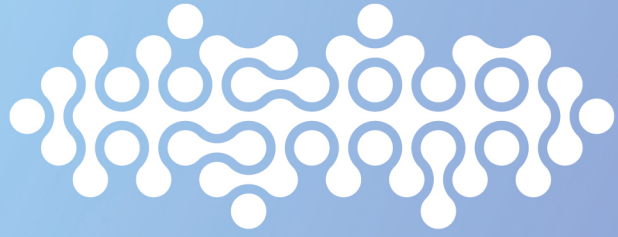
You now know project context files **work** — you saw the before/after.

You know it is **not perfect** — the benchmark anecdote.

But you do not yet know **why**.

In Act II, you will learn how these instructions are processed at the token level — why they influence but do not guarantee, why longer contexts degrade performance, and why the order of instructions matters.

For now: trust but verify.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Getting Started

5

Setup Checklist

```
# 1. Clone
git clone https://github.com/hs-aalen-software-engineering/locust.git
cd locust

# 2. Install dependencies
uv sync

# 3. Activate pre-commit hooks
pip install pre-commit && pre-commit install

# 4. Run the fast test subset
uv run pytest locust/test/test_env.py locust/test/test_tags.py \
    locust/test/test_users.py locust/test/test_util.py -x -q
```

If all tests pass: you are in a known-good state. **This is now hands-on** — follow along with the [Locust Exploration](#) exercise.

Create Your Branch + Diary

```
# Create your team's branch  
git checkout -b feature/<team-name>-<task>  
  
# Create your diary subfolder  
mkdir -p diary/feature/<team-name>-<task>
```

Your first diary entry: `001-onboarding.md`

Document: what you asked the AI, what you learned about the architecture, what the AI got right vs wrong.

Commit with: `git commit -m "[diary] 001 – Onboarding and exploration"`

Your First AI Onboarding Question

Open your AI tool in the Locust directory. Try:

"What is the high-level architecture of this project?"

Then compare the answer to what `CLAUDE.md` / `copilot-instructions.md` describes.

- Where does the AI add detail beyond the context file?
- Where does it get something wrong?
- What would you need to know to start contributing?

Do this now. This is the beginning of your exercise.

Full exercise: [Locust Exploration](#)

Next Session

Session 2 will cover:

- **Measuring success** — tests, coverage, benchmarks, Definition of Done
- **Exercise packages** — your team's specific targets
- **60 minutes of hands-on time** — write tests, refactor code, submit PRs

Before next session:

- Complete the [Locust Exploration](#) exercise
- Have your setup working: clone, tests passing, pre-commit hooks active



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

See You Next Session

GET YOUR SETUP WORKING!