



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Welcome

FROM COPY-PASTE TO
CRAFTED PROMPTS

Today's Agenda

1. **Feedback** on your Stack Overflow Challenge experience
2. **Three approaches** to the same challenge — copy-paste → structured → professional
3. **The AI Diary** — tracking your prompts as engineering artifacts
4. **GitHub Copilot & Claude Code** — two tools, three modes
5. **Challenge Round 2** — same task, new approach

Core insight: *The way you interact with an AI tool matters more than which tool you use.*

Recap: The Stack Overflow Challenge

In Lecture 1, you were given this task:

Your mission: *Recreate the Stack Overflow plot using Python + AI tools.*

- Find the data ***programmatically***
- Reproduce the chart: monthly question counts + rolling average
- ***Extend it:*** Overlay markers for key AI breakthroughs

Full exercise: [Stack Overflow Challenge — Round 1](#)

How Did It Go?

Quick show of hands:

- Who got a **working plot**?
- Who got **partially working** code?
- Who got **stuck** on data access?
- Who encountered **hallucinated API endpoints**?
- Who **read every line** of the AI-generated code?

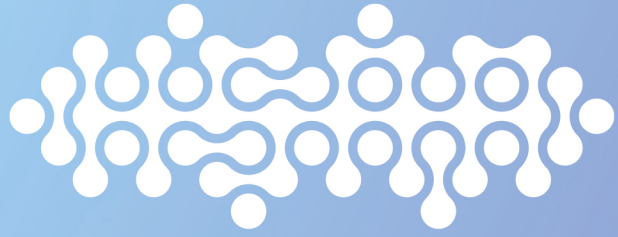
Feedback Time

Please fill out this short feedback quiz about your experience:

Stack Overflow Challenge — First Attempt Feedback

We will look at the aggregated results together.

Pay attention to the patterns — your struggles were likely shared by many peers.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Three Approaches to the Challenge

1

Approach 1: The Copy-Paste

What you do: Take the task description and paste it into the AI chat. Done.

Your mission: Recreate the Stack Overflow plot using Python + AI tools.

- Find the data programmatically (Stack Overflow Data Explorer, ...)
- Reproduce the chart: monthly question counts + rolling average
- Extend it: Overlay markers for key AI breakthroughs

Create a Python script that does all of this.

Or: paste a **screenshot of the slide** directly into the chat.

Why Are All Prompts in English?

LLMs are trained predominantly on English text (~90%). Research shows:

- **English prompts produce better reasoning** — even when the question is in another language ([Shi et al., 2022](#); [Etxaniz et al., 2023](#))
- **"Cross-Lingual-Thought Prompting"** — routing reasoning through English yields significant gains ([Huang et al., 2023](#))
- **Internal representations are English-aligned** — middle layers where reasoning happens operate in an English-centric space ([Wendler et al., 2024](#))

Does the model translate your German prompt internally? No literal translation — but its strongest reasoning pathways were shaped by English data.

Best practice: Prompt in English, request output in any language you need. The model generates German tokens directly — no hidden translation step.

What Happens

The AI generates a complete 50-100 line script. Looks impressive.

But then:

- **Hallucinated API endpoints** — plausible URLs that return 404
- **Hardcoded fake data** — chart looks great, numbers are made up
- **Monolithic structure** — everything in one function, untestable
- **Missing dependencies** — `ModuleNotFoundError` on first run

What follows: 10-15 rounds of *"This doesn't work because..."*

The 80/20 Problem

The first 80%:

- AI gets you to "looks like it works" in minutes
- Initial speed is seductive

The last 20%:

- Debugging, data issues, subtle bugs
- Takes hours, not minutes
- Each fix risks breaking other parts

Hallucination surface area: A broad prompt with no constraints gives the AI maximum freedom to generate plausible-sounding but wrong content.

Did You Understand the Code?

Honest question: When the AI generated 80 lines of code in seconds...

- Did you **read** it?
- Could you **explain** why it uses those API parameters?
- Could you **fix** the date parsing if it broke in a new way?

Using code you don't understand makes it **much harder to maintain** — even with AI tools that can help with future changes.

This is the crucial part. Not a rule — an **awareness**.

Approach 2: The Structured Breakdown

What you do: Spend 5-10 minutes thinking **before** touching the AI.

Identify distinct concerns:

1. **Data research** — Where does the data come from?
2. **Data access** — Fetching and caching
3. **Visualization** — Plotting with formatting
4. **Milestone data** — The 6 AI breakthrough dates
5. **Integration** — Tying it all together

Then write **multiple focused prompts** instead of one big one.

Experience Shapes Your Prompts

The example prompts that follow reflect someone with API and Python experience. **Your prompts will look different — and that's fine.**

Experienced developer:

"Fetch monthly SO counts using SEDE, return as pandas DataFrame with columns year_month, question_count"

Newer developer:

"I need monthly Stack Overflow question counts from 2008-2024. How can I get this data?"

The **strategy** matters (decompose, research first), not the specific terms you use.

Prompt 1 — Research

I need to get the monthly count of questions posted on Stack Overflow from 2008 to 2024. What are my options?

Consider: Stack Exchange API, SEDE, data dumps, web scraping, pre-existing datasets.

For each: pros, cons, rate limits, access requirements.

This prompt asks for **research**, not code.

You're using the AI as an **analyst** before using it as a coder.

Prompt 2 — Data Access Module

Write a Python module ``data_fetcher.py`` that fetches monthly SO question counts using SEDE.

Requirements:

- Return data as a pandas DataFrame
- Cache results locally as CSV
- Handle network errors with retry logic

Focused, single-responsibility task. Precise requirements leave less room for hallucination.

Being overly prescriptive can also hurt — the AI might find a better approach if you leave some freedom.

Prompt 4 — Milestone Data

The task says: *"Overlay markers for key AI breakthroughs from the timeline."*

These are **known data** from the course timeline:

- Transformer (June 2017) · ChatGPT (Nov 2022) · GPT-4 (Mar 2023)
- Cursor & AI IDEs (Aug 2023) · Claude Code (Feb 2025) · Copilot Agent Mode (Feb 2025)

With only 6 milestones, you could just **type them manually**. That's a valid engineering decision.

Not everything needs to be AI-generated. Knowing when *not* to use the AI is part of the skill.

What Is Different in Approach 2

- Each module is **self-contained** and testable independently
- Hallucinations are **isolated** — a bad API call is in `data_fetcher.py`, not everywhere
- **Research before implementation** consistently produces better results

Key insight: *Prompting is design.* How you decompose prompts mirrors how you decompose software. This is not a coincidence.

Could This Be One Prompt?

I need a Python project that visualizes monthly SO question counts with AI milestone markers.

Before writing code, first:

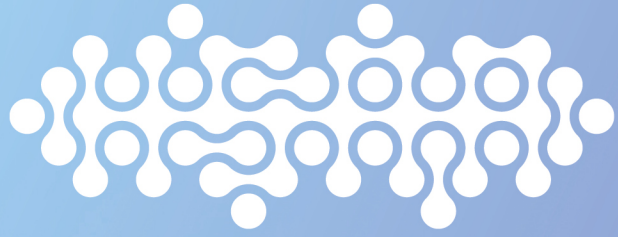
1. Research available data sources
2. Design a modular architecture
3. Define each module's interface

Then implement each module one at a time.

This encodes the **thinking process** as instructions. One prompt, but structured.

What you gain: Convenience. **What you lose:** Control over design decisions.

Preview: This is a stepping stone to **agentic workflows** (Act III).



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Why Track Your AI Interactions?

2

The Problem with Chat History

Built-in chat history (Copilot, ChatGPT) has **significant gaps**:

- **No artifact correspondence** — which prompt led to which code change?
- **No iteration tracking** — how many back-and-forth cycles did a task take?
- **No intent tracking** — what you *tried* to do vs. what you *asked* for
- **No cross-tool comparison** — histories in different places, different formats
- **Fragile storage** — sessions can be lost when VS Code restarts

Why Tracking Matters

Self-reflection. You are learning a new skill. Improvement requires conscious reflection. Without records, this is based on vague memory.

Future coursework. In Act III, you will analyze your own prompting strategies. You will need *data* — prompt counts, hallucination patterns, iteration trends. The diary is your dataset.

Professional evidence. By the end of this course, your diary is a portfolio of *how* you work with AI tools — a much more valuable signal than code alone.

Approach 3: The Professional

Before any prompting, you decide on a **professional development process**:

1. Set up a project with proper packaging — **git repo from the start**
2. Commit each meaningful milestone separately
3. Define requirements **as a document** before writing code
4. Write **tests before implementation**
5. Choose infrastructure tooling (`uv` , `gh`)
6. Track every AI interaction in a `diary/` **folder**

This embeds the **Software Development Lifecycle** into your prompting strategy.

Prompt 0 — Project Setup (Part 1)

Create a Python project called so-challenge using uv for dependency management. Separate data collection (data_fetcher.py) from visualization (plotter.py), with a module for milestones (milestones.py).

Set up pytest with test files for each module. Manage all dependencies through pyproject.toml. Include a README.md.

Create a diary/ folder for tracking AI interactions.

This is the **scaffolding** part — what files and structure to create. But the prompt continues...

Prompt 0 — Diary Format (Part 2)

For every prompt cycle, save the interaction record to diary/ as a numbered markdown file. Use this format:

```
# NNN — Short Title
**Date**: YYYY-MM-DD  **Tool**: [name]  **Model**: [name]
**Iterations**: [number]
## Prompt
**YYYY-MM-DD HH:MM**
[The full prompt text as given by the user.]

Save this setup prompt as diary/001-project-setup.md.
```

You must define the format — or the AI invents a different one each time.

Prompt 0 — Constraints (Part 3)

Do NOT implement any logic yet – just create the project structure, configuration, and empty module files with docstrings describing their purpose.

Initialize a git repository and create an initial commit.
Then create a GitHub repository using the gh CLI and push.

Three critical elements:

- **"Do NOT implement"** — controls what the AI does *not* do
- **Git from the start** — scaffolding gets its own commit before any code
- **Intent over layout** — describes *what you want*, not an ASCII directory tree

Prompt 1 — Requirements Document

Write a brief requirements specification for this project and save it to docs/requirements.md.

Include:

- Functional requirements (data source, date range, plot type)
- Non-functional requirements (caching, error handling)
- Acceptance criteria for each requirement

Save the diary entry and commit everything together with a meaningful commit message.

This produces a **document, not code** — and commits it immediately with its diary entry.

Prompt 2 — Tests First (Red Phase)

Write pytest tests for `data_fetcher.py`. The module should:

- Fetch monthly SO question counts (2008-2024)
- Return a pandas DataFrame
- Cache results locally as CSV
- Handle network errors gracefully

Write the tests BEFORE the implementation. Use mocks for network calls.

Save the diary entry and commit with a meaningful commit message.

These tests will **fail** — that's the point. They define the expected behavior *before* implementation.

Prompt 3 — Implementation (Green Phase)

```
Implement data_fetcher.py to pass the tests. Do not modify  
the tests. Run the tests to confirm they pass.
```

```
Save the diary entry and commit with a meaningful commit  
message describing what was implemented.
```

No need to paste the test code — you're in the same session, the AI already has context.

"Proper commit message" — the AI derives it from changes, not a dictated format. In a real project, commit templates and conventions can be encoded in project instruction files.

Repeat: Red → Green → Commit

Apply the same pattern for each module:

plotter.py : tests → implementation → commit

milestones.py : tests → implementation → commit

main.py : integration → commit

Each cycle produces two commits: tests first, then implementation. Each commit includes its corresponding diary entry.

Or with an **agentic tool** like Claude Code, you can compress the cycle:

"Implement data_fetcher with tests. If tests pass, commit and move to plotter."

The Resulting Git History

```
* f4e2a1b Integrate main.py and update README
* c8d3b7e Implement milestones module
* 91a0f5c Add tests for milestones module
* 7b2e4d9 Implement plotter module
* 3f8c1a6 Add tests for plotter module
* a5d9e2f Implement data_fetcher module
* e1c7b3d Add tests for data_fetcher module
* 6d4f8a1 Add requirements specification
* 9c3e8f1 Initial project scaffolding
```

Each commit tells a story. Each includes a diary entry in `diary/`.

`git show` on any commit → code changes + the prompt that produced them.

What Makes Approach 3 Different

- **Tests as hallucination detectors** — hallucinated code fails immediately
- **Infrastructure matters** — `uv sync` and everything works, no "works on my machine"
- **SDLC embedded in prompting** — requirements → tests → implementation → commit
- **The AI manages process, not just code** — requirements docs, commit messages, diary entries

Same experience caveat: if `uv`, `pytest`, and TDD are new to you, start simpler. The **core principle** is the same: let engineering process guide your prompting.

The Repetition Problem

Notice two annoyances:

1. **Prompt 0** contains a long diary format specification
2. **Every subsequent prompt** includes "save the diary entry and commit"

Within one session, the AI remembers. **But tomorrow? New session = no memory.**

This isn't just a diary problem. Think about all the conventions you repeat:

- Commit message format
- Testing requirements
- Coding standards
- Library preferences

Project Instruction Files

Text files the AI reads **automatically** at the start of every session:

Tool	File
Claude Code	CLAUDE.md
GitHub Copilot	.github/copilot-instructions.md
Cursor	.cursorrules

Same concept, different filenames. **Encode your conventions once, the tool applies them always.**

We will set these up in Act III. For now: understand **why** they exist.

CLAUDE.md Example

```
# Project Instructions
```

```
## AI Diary
```

```
For every prompt cycle, save a diary entry to diary/  
as a numbered markdown file with: date, tool, model,  
iterations, and full prompt text with timestamps.  
Always commit diary entries with the code changes.
```

```
## Conventions
```

- Run tests before committing implementation code
- Use meaningful commit messages derived from changes
- Do not modify existing tests when implementing

With this file, you never mention the diary in any prompt. **Every session just knows.**

Discussion: Your CLAUDE.md

If you were setting up a CLAUDE.md for your project, what would you put in it?

Think about:

- What do you find yourself repeating in prompts?
- What conventions does your team follow?
- What should the AI *never* do?
- What should the AI *always* do?

Take 2 minutes. Share one idea with the class.

Comparing the Three Approaches

Aspect	Copy-Paste	Structured	Professional
Setup	None	Minutes thinking	Significant
Prompts	1 + fix-ups	4-6 focused	8-12 systematic
Hallucination risk	High	Medium	Low (tests!)
Debugging	Hard	Medium	Easy
Reproducibility	Low	Medium	High

Time estimates are **experience-dependent** — the relative pattern holds, not the absolute numbers.

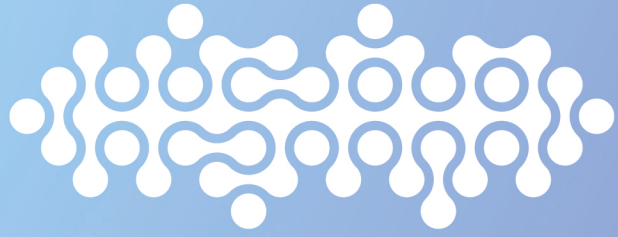
Key Messages

None of these approaches is "wrong." They represent a progression.

- Approach 1: fine for quick experiments
- Approach 2: appropriate for most tasks
- Approach 3: when quality and maintainability matter

This is not about prompting tricks. It's about applying **software engineering thinking** to AI-assisted development.

The AI doesn't know what "good" looks like for your project. You encode quality through structure, requirements, and tests.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The AI Diary

3

Diary Entry Format

```
# 004 – Data Fetcher Implementation
```

```
**Date**: 2026-03-23
```

```
**Tool**: Claude Code
```

```
**Model**: Claude Opus 4.6
```

```
**Iterations**: 2
```

```
## Prompt
```

```
**2026-03-23 10:25**
```

```
Implement data_fetcher.py to pass the tests...
```

```
**2026-03-23 10:32**
```

```
The test test_retries_on_network_error is failing...
```

Full prompts with timestamps. No summaries. Raw record of the conversation.

Project Structure with Diary

```
so-challenge/  
├─ src/so_challenge/  
│   ├─ data_fetcher.py  
│   ├─ plotter.py  
│   └─ milestones.py  
├─ tests/  
├─ docs/requirements.md  
├─ diary/  
│   ├─ 001-project-setup.md  
│   ├─ 002-requirements.md  
│   ├─ 003-data-fetcher-tests.md  
│   ├─ 004-data-fetcher-impl.md  
│   └─ ...  
└─ pyproject.toml
```

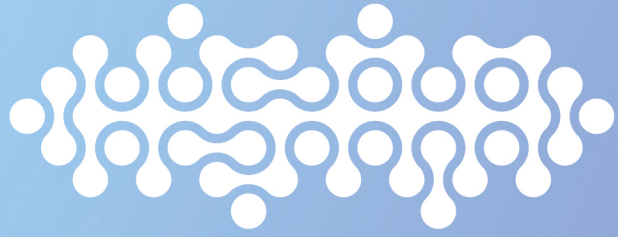
Each diary entry committed **alongside** the code it produced.

Why Version-Control Your Prompts?

In a production project, you probably wouldn't do this. But **this is a course about building AI competence.**

Prompts are engineering artifacts. If you wouldn't throw away your source code, why throw away the instructions that created it?

- **See which prompt created which code** — one `git diff` tells the story
- **Compare prompts over time** — did you improve between week 2 and week 10?
- **Compare across models** — same prompt to Copilot vs. Claude, easy to diff
- **Share and discuss** — prompts provide context that code alone cannot



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

GitHub Copilot & Claude Code

4

Two AI Tools in VS Code

In Agent mode, **both tools** can: read/write files, run commands, execute tests, make git commits, use `gh` CLI. Core capabilities have converged.

Aspect	GitHub Copilot	Claude Code
Interface	Chat panel + inline completions	Terminal panel + standalone CLI
Models	GPT-4o, Claude Sonnet, Gemini	Claude Opus, Claude Sonnet
Context window	128K-200K tokens	Up to 1M tokens (Opus)
Instructions file	<code>.github/copilot-instructions.md</code>	<code>CLAUDE.md</code>
Student pricing	Free (GitHub Education)	Free tier (limited daily)

Interaction Modes

Mode	GitHub Copilot	Claude Code
Ask	Ask — answers, no edits	Ask — answers, no edits
Agent / Edit	Agent — edits, you review	Edit automatically
Plan	Plan via <code>/plan</code>	Plan — design first

The names differ, the concept is the same: **how much autonomy do you give the AI?**

- **Ask:** AI is a consultant. It advises, you execute.
- **Agent/Edit:** AI is a collaborator. It changes, you review.
- **Plan:** AI is an architect. It designs, you approve, then it builds.

When to Use Which

Both tools can handle almost any task in Agent mode. Some starting observations:

- **Copilot's inline completions** (outside Agent mode) are great for in-flow edits — no equivalent in Claude Code
- **Copilot's model flexibility** — switch between GPT-4o, Claude Sonnet, Gemini in one session
- **Claude Code's context window** — up to 1M tokens, advantage for large codebases
- **Claude Code's standalone CLI** — works outside VS Code, useful for scripting and CI/CD

Try both and compare. Document your preferences in the AI Diary — these are exactly the kind of observations the diary is for.

Configuring Permissions

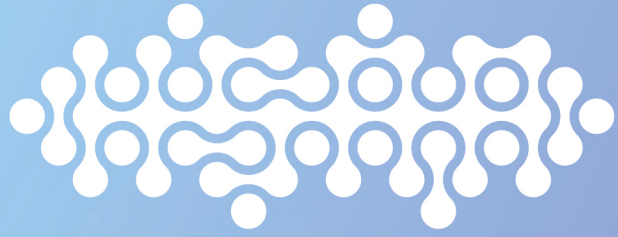
Both tools ask before running commands. For repeated workflows, configure auto-approve:

Claude Code — `~/.claude/settings.json`:

```
{ "permissions": { "allow": [
  "Edit", "Write", "Bash(git:*)", "Bash(uv:*)"
] } }
```

Copilot — VS Code trust model: click "Allow" or "Always Allow" per command pattern.

Best practice: Start with defaults, add permissions as you discover what you run repeatedly.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Challenge Round 2

5

Stack Overflow Challenge — Round 2

Same task. Your approach should not be the same.

Use **Approach 3** — the professional SDLC workflow with integrated diary.

Deliverables:

- A GitHub repo with clean commit history (code + diary entries per commit)
- A `diary/` folder with numbered entries
- The generated Stack Overflow trend plot

Reference repo: github.com/l3gj0n/so-challenge

Full exercise: [Round 2](#)

Focus on Process, Not Just Result

A **working plot** with no commit history and no diary is worth less than a **partially working plot** with a clean git log and complete diary entries.

We are assessing **how** you work, not just **what** you produce.

Timeline: Start during this session if time permits. Complete by Lecture 3.

Post-challenge feedback will be collected after you complete Round 2 — we will compare your experience across both attempts at the start of Lecture 3.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

THANK YOU

Your feedback is welcome.