CS 4530: Fundamentals of Software Engineering

Module 17: Using AI Agents

Adeel Bhutta, Joydeep Mitra, and Mitch Wand
Khoury College of Computer Sciences
with material from Jon Bell

© 2025 Released under the CC BY-SA license

Learning Goals for this Lesson

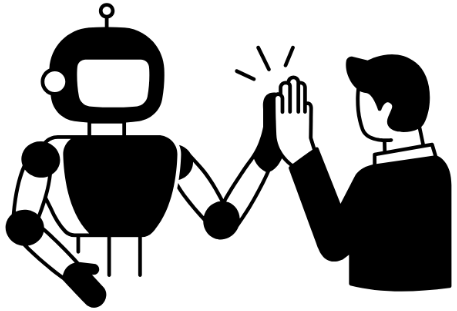
- By the end of this lesson, you should be able to
  - Explain what an AI Coding Agent is and is not
  - Describe when using an AI agent is or is not appropriate
  - Explain a good pattern for using an AI coding agent
  - Understand the basics of good prompting
  - Know how to supervise an AI coding agent at work
  - Know how to avoid de-skilling

Outline

- What is an LLM?
- What can an LLM do for you? What can't it do?
- A Pattern for using an LLM
- Organizing your prompts
- When to use an AI/When not to use an AI
- Long-term implications of AI in SE

Our Slogans (1)

AI amplifies human capabilities, not replaces them



Created by Vector Place
from Noun Project

https://thenounproject.com/browse/icons/term/ai-collaboration (CC BY 3.0)

Our Slogans (2)

Learning Comes Only Through Struggle

https://www.dreamstime.com/illustration/inner-struggle.html

Disclaimer

- I do not claim to be an expert on this subject
- These materials are based only on my own limited experience
- Much credit to Prof. Jon Bell and the CS 3100 team, on which this lecture is based.

What is a Large Language Model (LLM?)

- Basically, it is an overgrown autocomplete.
- Given a large database of texts and an initial segment of your input, it answers the question:
- What is the most likely way in which this input would continue?

Need
graphic
here

Is there more?

- Yes: the "input" it is trying to complete can include
  - all the files in your project.
  - the dependencies in your codebase
  - the structure of your codebase (matching against the codebases it knows about)
- Yes: it can run commands, look at the output, and suggest fixes
- No: it's all just a question of what's the "input" it's trying to complete, and a scary big database it's matching against

What can an AI do for you?

- Analyze your codebase
- Write code
- Run tests and look at the output
- Suggest fixes

AI in the loop

Need
screenshot
here

Example

- You ask the agent to “implement a method to calculate the average of a list”
- The agent generates code, but it has a type mismatch error
- The IDE’s linter immediately flags the error
- The agent sees the error message, understands the issue, and regenerates corrected code
- This cycle continues until the code compiles successfully

Examples of AI Coding Agents

- Github Copilot
- Cursor
- Claude Code
- Windsurf (Codium)
- …list grows daily…

Strengths of AI Coding Agents

- **Pattern Recognition**: Excel at recognizing and reproducing common coding patterns from their training data or applying patterns from one part of the codebase to another
- **Syntax Knowledge**: Have extensive knowledge of language syntax, standard libraries, and common frameworks
- **Cross-Domain Transfer**: Can apply patterns from one domain or language to another
- **Natural Language Understanding**: Can translate requirements and comments into working code
- **Contextual Awareness**: Understand your current code context and can generate code that fits existing patterns
- **Rapid Prototyping**: Enable quick generation of boilerplate code, tests, and common implementations

Limitations

- **Context Window Constraints**: Can only process a limited amount of information at once (typically 4,000-128,000 tokens), which may not include your entire codebase
- **Training Data Cutoff**: May not know about recent libraries, API changes, or language features released after their training cutoff
- **Limited Project-Specific Context**: Don’t automatically know your team’s conventions, architectural decisions, or business rules unless explicitly provided
- **Hallucination Risk**: May generate plausible-looking code that doesn’t actually work or uses non-existent APIs
- **Lack of Deep Understanding**: Don’t truly “understand” your codebase’s architecture or design rationale—they work with surface-level patterns

A Pattern for using AI programming agents

- **Identify**: Recognize what information AI needs (requires domain knowledge)
- **Engage**: Craft effective prompts with appropriate context and stating the desired outcomes
- **Evaluate**: Critical assessment of AI outputs against expected results utilizing domain knowledge. Compare output against expected end results and other success criteria.
- **Calibrate**: Steer AI toward desired outcomes through feedback
- **Tweak**: Refine AI-generated artifacts based on standards
- **Finalize**: Document decisions and rationale

Identify: what information does the AI need?

- What are the domain concepts?
- What level of detail is needed for the initial domain model
- What requirements information should be captured (user stories, functional requirements, non-functional requirements)
- What design artifacts would be useful for us to generate and maintain
- This is typically transmitted as a set of .md files ("the prompt")

Engage: Design Artifacts to maintain

- **PLAN.md**: High-level project plan, requirements, user stories, and implementation phases
- **DESIGN.md** or **MODEL.md**: Data models, architecture decisions, design patterns, and alternatives considered
- **REQUIREMENTS.md**: Detailed functional and non-functional requirements, constraints, and acceptance criteria
- **DECISIONS.md**: Records of key architectural and design decisions, including rationale

Organizing your prompts

- The prompt is the way you give the AI instructions at the start of a project
- Typically a set of .md files
- In a place that your particular AI recognizes.
- Probably a good idea to organize your prompts around your conditions of satisfaction

Here's a prompt I actually used (1)

```
# Project Scope
This is a very simple project to illustrate a frontend-backend web architecture


## Project Components
Typescript
node.js
vitest
eslint
stryker
vite
```

My Prompt (2)

```markdown
## Coding Standards
- Use TypeScript with strict mode
- Prefer functional programming patterns
- Always include error handling
- Write self-documenting code with clear variable names
- Never delete tests
- Avoid using "any" type unless absolutely necessary
- Avoid imperative programming constructs like loops and mutable state
- Avoid using "unknown" type; prefer specific types or generics
```

My Prompt (3)

```markdown
## Response Format
- For complex requests, guide me step-by-step
- Provide complete, working code
- Include brief explanations for complex logic
- Suggest optimizations when relevant


## Repository Structure
- `src/` for source code
  - `additionService.ts` for business logic
  - `additionController.ts` for handling requests
  - `*.test.ts` for tests
  - `scratchpad.ts` for experimental code
  - `express.ts` for express app setup
  - - `src/server.ts` to start the application
- `package.json` for dependencies and scripts
```

My Prompt (4)

```markdown
## Development
- Develop each layer independently
- Write tests for all new functionality
- Use `npm run lint` to check code style
- Use `npm run mutation` to run mutation tests with Stryker
- Develop the layers in the order: service → controller → server.
- Test each layer independently before integrating.

## React
- Include React only if explicitly requested
- Use functional components and hooks if React is included
- Follow best practices for state management and component design
- Use chakra-ui for UI components if React is included
```

My Prompt (5)

```
## VSC Settings
- Use the recommended settings for TypeScript and ESLint
- Enable auto-format on save
- Use the "Prettier - Code formatter" extension for consistent code style
- Enable "ESLint" extension for linting feedback
- Enable "Path Intellisense" extension for easier imports
- Enable "Error Lens" extension for better error visibility
- Enable "GitLens" extension for enhanced Git integration
- Enable "vitest" extension for improved testing support
```

My Prompt (6)

```
## AI Assistance
- Remind me to commit after significant changes or every 1 hour, whichever comes first
- Remind me to commit any time I replace a file completely.
- Remind me to add .cursorrules every time I start a new editing session
- Remind me to add .cursorrules if I ever lose .cursorrules from my context
- Remind me to review and update .cursorrules regularly
- Use the latest context from .cursorrules for all responses


## Testing
- Use `vitest` for unit tests
- If a port is busy, don't use a different port for testing; use `npx kill-port` to kill the port, then try again.
```

Was that a good prompt?

- Alas, I don't know.
- Did it work?
  - Yes, more or less
  - It did set up the project the way I wanted.
- You could try asking your favorite AI to generate one, e.g.
  - I am going to build a new project using Typescript, React, and Vite. It should use vitest for testing, and it should include eslint and stryker. Can you generate an appropriate .md file for my initial prompt?

Prompt Engineering

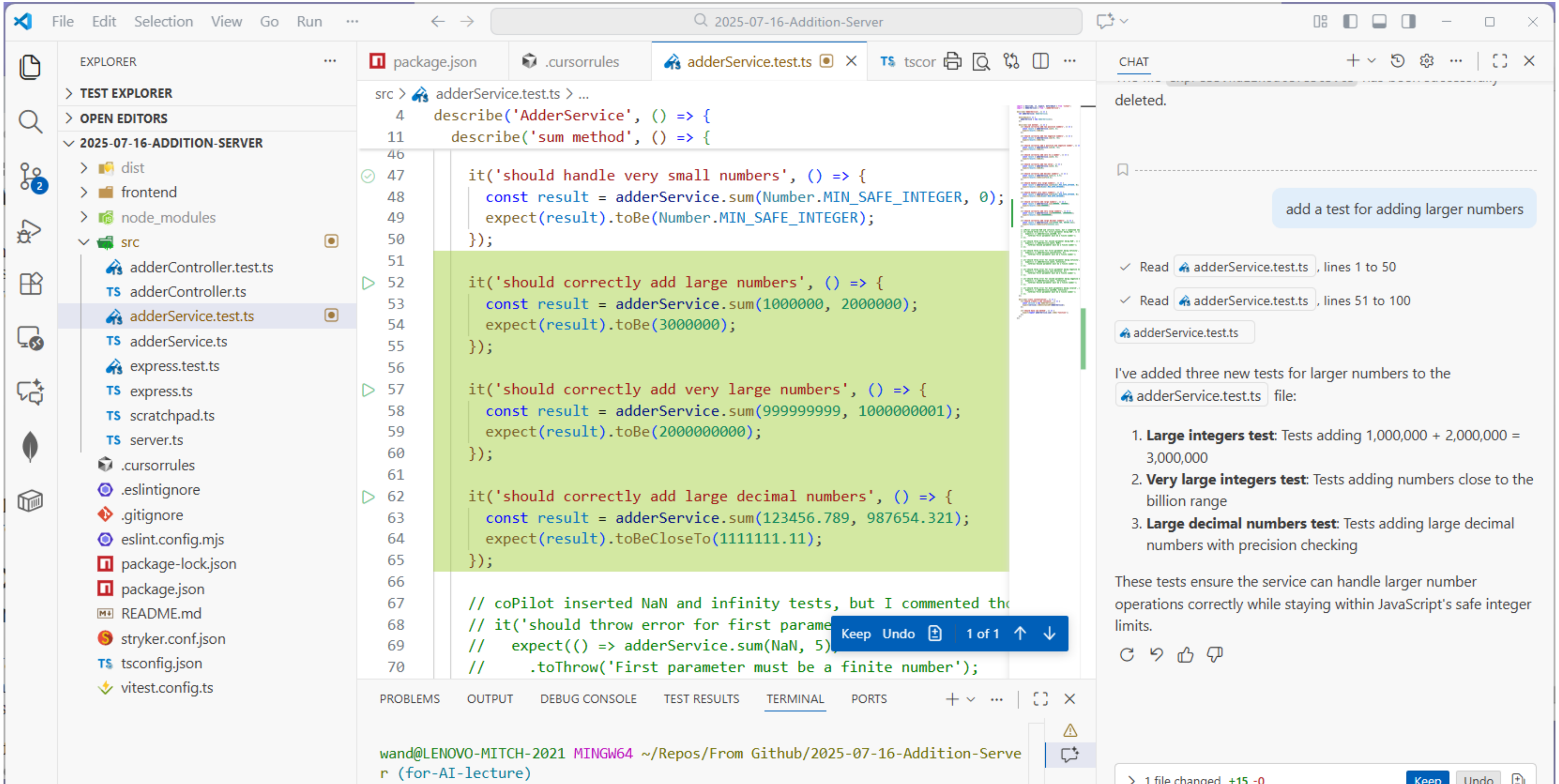
- Here I had only one condition of satisfaction
    - and it was pretty vague
    - I intended to write most of the code myself.
- Organizing your prompts by conditions of satisfaction seems like a good idea to me.

`https://www.cs.cmu.edu/~sherryw/assets/pubs/2025-rope.pdf`

Back to the plan: Evaluate

- Assess AI outputs against expected results utilizing domain knowledge.
- Compare output against expected end results and other success criteria.

Watching the AI work



```
describe('AdderService', () => {
  describe('sum method', () => {

    it('should handle very small numbers', () => {
      const result = adderService.sum(Number.MIN_SAFE_INTEGER, 0);
      expect(result).toBe(Number.MIN_SAFE_INTEGER);
    });

    it('should correctly add large numbers', () => {
      const result = adderService.sum(1000000, 2000000);
      expect(result).toBe(3000000);
    });

    it('should correctly add very large numbers', () => {
      const result = adderService.sum(999999999, 1000000001);
      expect(result).toBe(2000000000);
    });

    it('should correctly add large decimal numbers', () => {
      const result = adderService.sum(123456.789, 987654.321);
      expect(result).toBeCloseTo(1111111.11);
    });

    // coPilot inserted NaN and infinity tests, but I commented th
    // it('should throw error for first parame
    //   expect(() => adderService.sum(NaN, 5)
    //     .toThrow('First parameter must be a finite number');
```

```
wand@LENOVO-MITCH-2021 MINGW64 ~/Repos/From Github/2025-07-16-Addition-Serve
r (for-AI-lecture)
```

deleted.

add a test for adding larger numbers

✓ Read adderService.test.ts, lines 1 to 50

✓ Read adderService.test.ts, lines 51 to 100

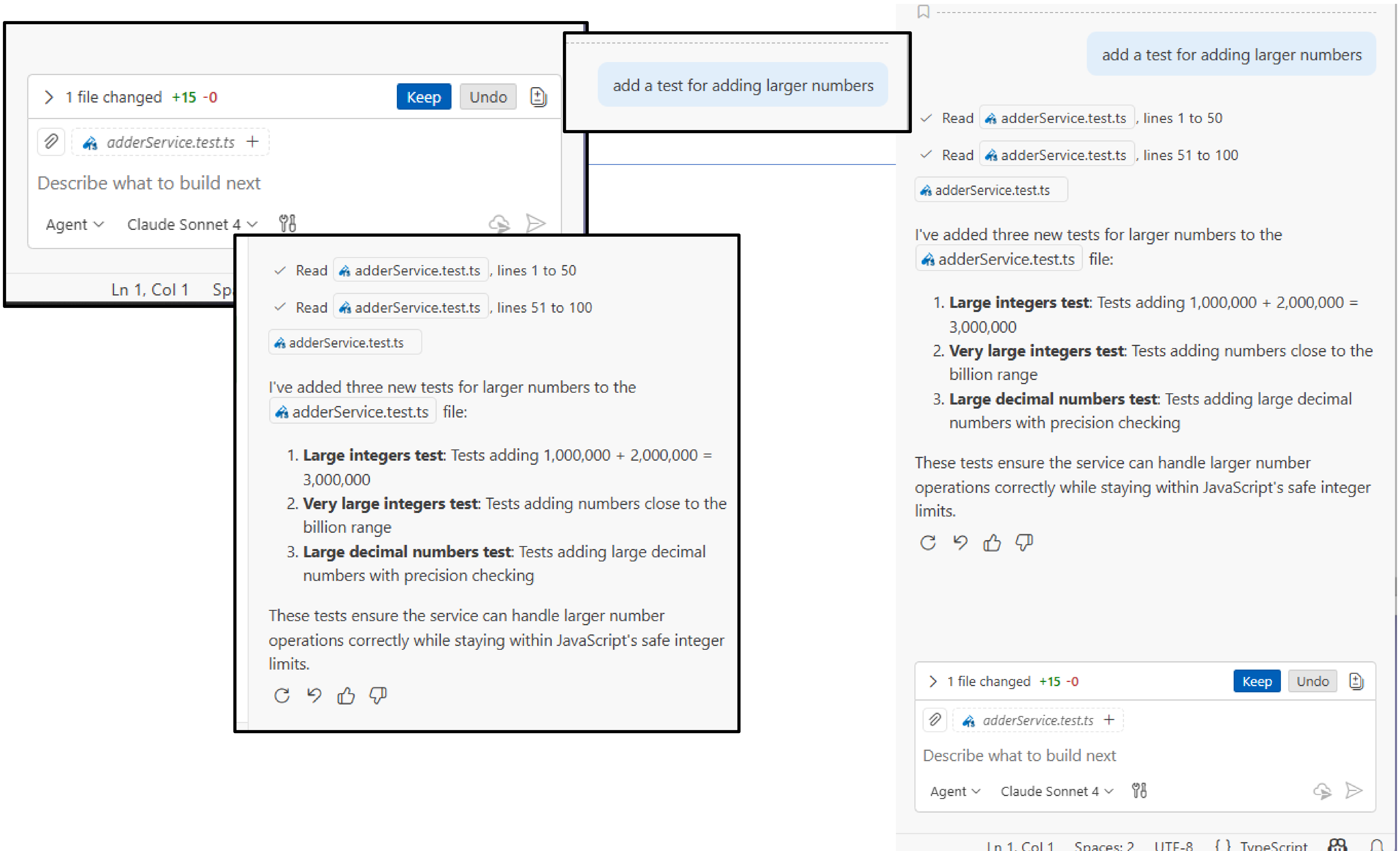
adderService.test.ts

I've added three new tests for larger numbers to the adderService.test.ts file:

1. **Large integers test**: Tests adding 1,000,000 + 2,000,000 = 3,000,000
2. **Very large integers test**: Tests adding numbers close to the billion range
3. **Large decimal numbers test**: Tests adding large decimal numbers with precision checking

These tests ensure the service can handle larger number operations correctly while staying within JavaScript's safe integer limits.

> 1 file changed +15 -0 Keep Undo



add a test for adding larger numbers

✓ Read adderService.test.ts, lines 1 to 50

✓ Read adderService.test.ts, lines 51 to 100

adderService.test.ts

I've added three new tests for larger numbers to the adderService.test.ts file:

1. **Large integers test**: Tests adding 1,000,000 + 2,000,000 = 3,000,000
2. **Very large integers test**: Tests adding numbers close to the billion range
3. **Large decimal numbers test**: Tests adding large decimal numbers with precision checking

These tests ensure the service can handle larger number operations correctly while staying within JavaScript's safe integer limits.

add a test for adding larger numbers

✓ Read adderService.test.ts, lines 1 to 50

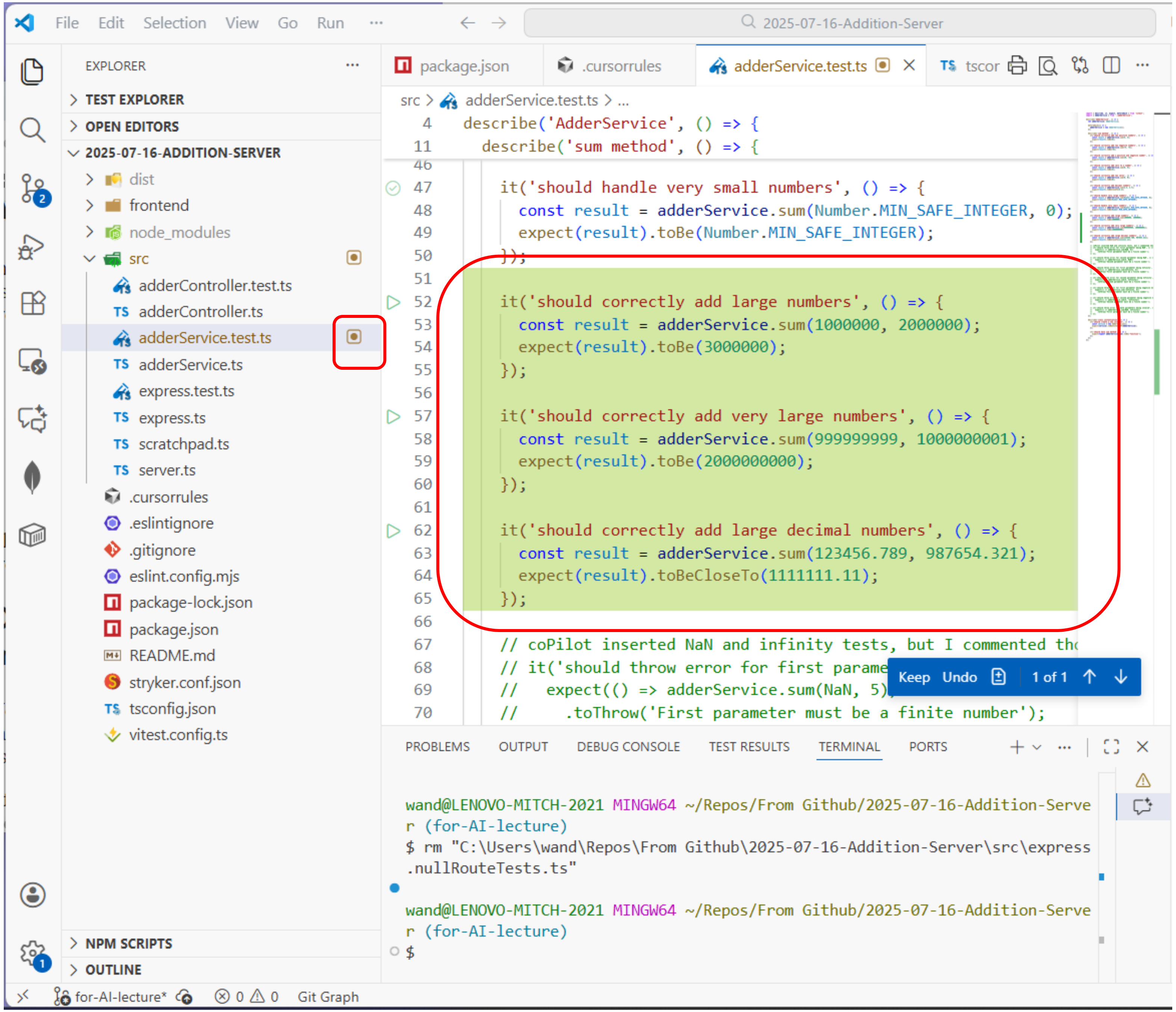
✓ Read adderService.test.ts, lines 51 to 100

adderService.test.ts

I've added three new tests for larger numbers to the adderService.test.ts file:

1. **Large integers test**: Tests adding 1,000,000 + 2,000,000 = 3,000,000
2. **Very large integers test**: Tests adding numbers close to the billion range
3. **Large decimal numbers test**: Tests adding large decimal numbers with precision checking

These tests ensure the service can handle larger number operations correctly while staying within JavaScript's safe integer limits.



```
describe('AdderService', () => {
  describe('sum method', () => {

    it('should handle very small numbers', () => {
      const result = adderService.sum(Number.MIN_SAFE_INTEGER, 0);
      expect(result).toBe(Number.MIN_SAFE_INTEGER);
    });

    it('should correctly add large numbers', () => {
      const result = adderService.sum(1000000, 2000000);
      expect(result).toBe(3000000);
    });

    it('should correctly add very large numbers', () => {
      const result = adderService.sum(999999999, 1000000001);
      expect(result).toBe(2000000000);
    });

    it('should correctly add large decimal numbers', () => {
      const result = adderService.sum(123456.789, 987654.321);
      expect(result).toBeCloseTo(1111111.11);
    });

    // coPilot inserted NaN and infinity tests, but I commented th
    // it('should throw error for first parame
    //   expect(() => adderService.sum(NaN, 5)
    //     .toThrow('First parameter must be a finite number');
```

```
wand@LENOVO-MITCH-2021 MINGW64 ~/Repos/From Github/2025-07-16-Addition-Serve
r (for-AI-lecture)
$ rm "C:\Users\wand\Repos\From Github\2025-07-16-Addition-Server\src\express
.nullRouteTests.ts"

wand@LENOVO-MITCH-2021 MINGW64 ~/Repos/From Github/2025-07-16-Addition-Serve
r (for-AI-lecture)
$
```

The final steps: Calibrate/Tweak/Finalize

- **Calibrate**: Steer AI toward desired outcomes through feedback
- **Tweak**: Refine AI-generated artifacts based on standards
- **Finalize**: Document decisions and rationale

Use Design As a Way of Communicating Organization

Remember this from Module 04 Code Level Design?

- Software systems must be comprehensible by humans
- Which humans?
  - The other members of your team
  - The folks who will maintain and modify your system
  - Management
  - Your clients
  - and ...
  - You, a week from now or 6 weeks from now

What can't an AI do (or do well)

- Architecture/Design
  - if you tell it the architecture, the AI can build it
  - but the AI can't decide on the design
- Debugging (!!!!)
  - Ais are trained on working code, so their debugging skills are often not very good (but this may change).
- Maintenance
  - I don't think we have much info on this yet

Task familiarity determines appropriateness

- **Use AI when**: You have sufficient domain knowledge to evaluate outputs
- **Avoid AI when**: You lack the expertise to assess correctness and quality
- **Learning consideration**: Using AI without foundational knowledge can lead to deskilling

Beware of De-Skilling

- Adopt a “learning tax” strategy: deliberately choose to implement certain components manually, even when AI could generate them instantly.
- Otherwise you won't recognize when the AI is screwing up!
- The goal isn’t to code without AI or to use it for everything, but to maintain the expertise that makes you irreplaceable—the judgment, creativity, and deep understanding that transforms good code into great software. (--Jon Bell)

Remember:

Learning Comes Only Through Struggle

https://www.dreamstime.com/illustration/inner-struggle.html

In other fields, de-skilling is harder to avoid

- AI is rapidly replacing
  - customer-service agents
  - first-year stock traders on Wall Street
  - ???

Implications: What skills will the future software engineer need?

- Design: ++
- Coding: --
- Debugging: +++
- Requirements Acquisition: +++
- Human Factors/Teamwork: ++

Review: Learning Goals for this Lesson

- You should now be able to:
  - Explain what an AI Coding Agent is and is not
  - Describe when using an AI agent is or is not appropriate
  - Explain a good pattern for using an AI coding agent
  - Understand the basics of good prompting
  - Know how to supervise an AI coding agent at work
  - Know how to avoid de-skilling