CS 4530: Fundamentals of Software Engineering
Module 16: Coding With AI Agents

Adeel Bhutta, Rob Simmons, and Mitch Wand

Khoury College of Computer Sciences

based on material from Jon Bell

This Is Just the Beginning of Our Conversation

- This topic is unlike anything covered so far:
  - The technology is evolving faster than any textbook can capture
  - There is genuine disagreement among experts about best practices
  - The hype is real—and so are the concerns
  - Your professors are learning alongside you, modeling the best practices we teach

Disclaimer

- I do not claim to be an expert on this subject
- Much credit to Prof. Jon Bell and the CS 3100 team, on which this lecture is based.
- I also had extensive conversations with claude.ai, who made good suggestions about restructuring the section on prompting. I'll credit these when we get to them.

Learning Goals for this Lesson

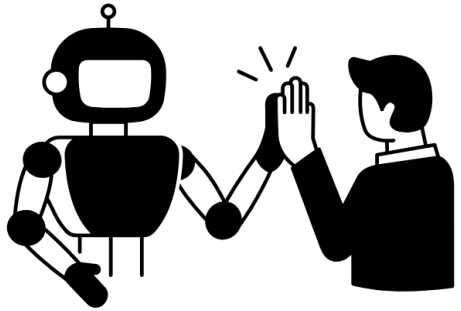
- By the end of this lesson, you should be able to
  - Explain what an AI Coding Assistant is and is not
  - Describe how to determine when using an AI assistant is or is not appropriate
  - Explain the 6-step Human-AI Workflow pattern
  - Explain the Plan-First pattern
  - Explain the Prompt-Contract pattern
  - Know how to avoid de-skilling

Outline

1. What is an LLM? What is an AI Coding Assistant?
   Why is context so important?
2. A 6-step Pattern for Human-AI Workflow
3. Patterns and Myths for Effective Prompting
4. A few words about AI traps

Our Slogan:

AI amplifies human capabilities; it doesn't replace them

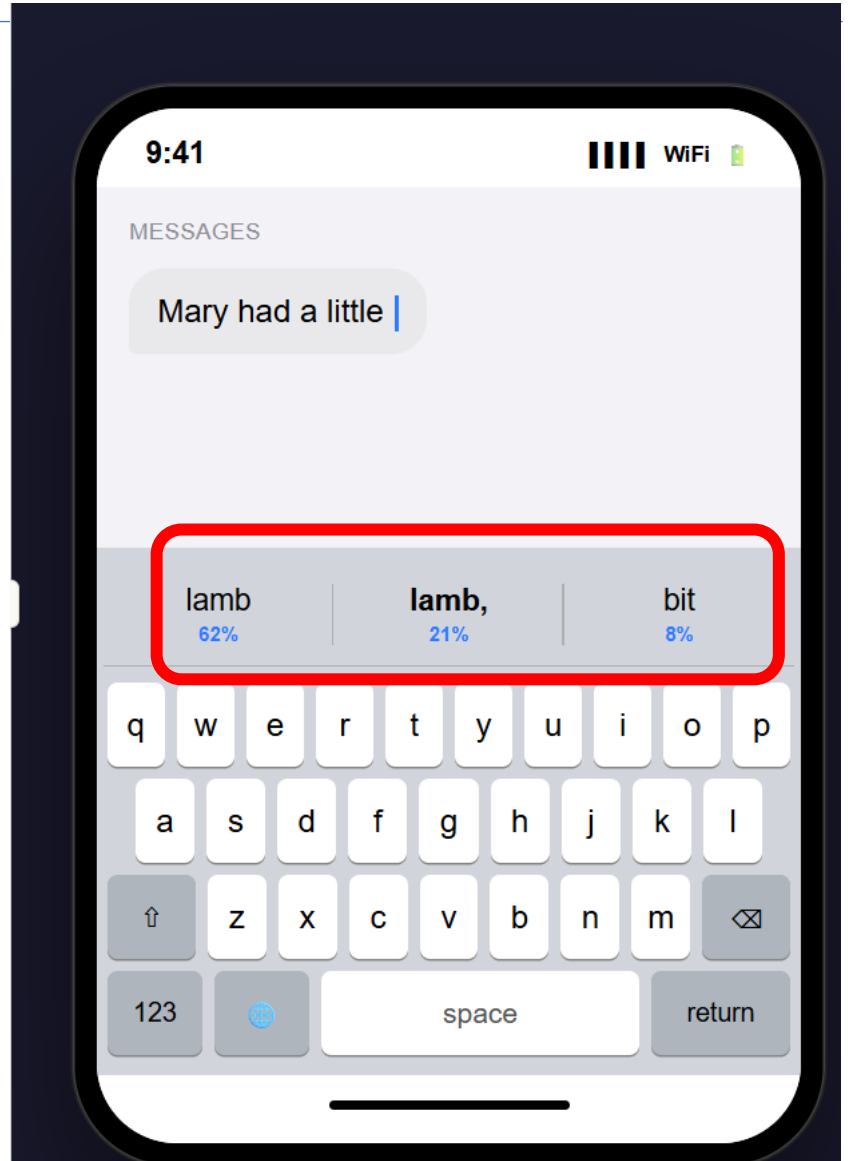


Created by Vector Place
from Noun Project

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

1. 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?
- More accurately: it produces a probability distribution of likely continuations, and draws from that distribution (hence non-deterministic)



Why this matters

- Explains hallucination: model predicts plausible-looking text, not necessarily true text
- Explains why context matters: predictions depend on what came before
- Explains strengths: patterns in code are highly predictable!

What is an AI Coding Assistant?

- An AI coding assistant typically consists of an LLM
  - trained on a large base of running code
  - operating in a loop with an IDE
    - it can add artifacts inside the IDE to its context
    - it can operate on your code (with your permission)

AI Coding Assistants: Strengths

- **Strengths:**
  - Pattern recognition: recognizes and reproduces common coding patterns across millions of codebases
  - Syntax knowledge: extensive knowledge of languages, libraries, and frameworks
  - Cross-domain transfer: can apply patterns from one language to another
  - Rapid prototyping: generates boilerplate, tests, and common implementations quickly
  - AI in-the-loop: can run tests and evaluate the results
    - did it work? if not it can try something else

AI Coding Assistants: Limitations

- **Limitations:**
  - Context window: can only see ~100K tokens at once — may miss parts of large codebases
  - Generates code based on patterns, not execution results
  - Hallucination: may generate plausible-looking code that doesn’t actually work, or doesn't satisfy your project constraints.
- *Think of it as a well-read junior developer who has seen millions of codebases — but has never run your specific project.*

Context Is Everything

- **Rich context → High-quality output**
- **Why IDE integration matters:**
  - Copilot sees your open files → more context
  - Claude Code reads your whole codebase → even more context
  - More relevant context = better predictions

The Secret Sauce: Static Analysis Powers Context

- How AI coding assistants automatically find the right context:
  - AST parsing: understands code structure, not just text
  - Semantic indexing: knows what symbols mean, where they're defined
  - Call graphs: traces which functions call which
  - Type information: knows the types flowing through your code
- When your cursor is in a method, the tool KNOWS the class you're in, the interfaces it implements, the types of parameters, and available methods — and sends EXACTLY this to the LLM, not your whole codebase.

Static Analysis Can't Find Everything

- What tools can auto-find: related files, type definitions, call graphs, similar patterns, test files
- **What tools cannot auto-find: why a design was chosen, undocumented requirements, knowledge in your head**
- *This is why DESIGN.md and documentation matter — it's context the tool CAN find.*

2. A 6-Step Human-AI Collaboration Workflow

- Identify — recognize what information AI needs
- Engage — craft effective prompts with context
- Evaluate — critically assess AI outputs against your success criteria
- Calibrate — steer AI toward desired outcomes through feedback
- Tweak — manually refine generated artifacts
- Finalize — document decisions and rationale

- Based on Google research studying 21 expert developers working with AI
- This is not a rigid sequence — steps may overlap and you will often loop back.

Identify and Engage: 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?

How much context you need to provide depends on three things:

1. **Your understanding:** Experts know exactly what context matters; novices may not know what to include. You can only identify relevant context for things you understand.
2. **Model capabilities:** Frontier models infer much more from less. GPT-3.5 needed explicit instructions; Claude Opus can often infer intent.
3. **Tool capabilities:** Basic Copilot sees only your open files. Cursor and Claude Code index your entire codebase and find files automatically.

- When all three are HIGH: just describe intent — the system figures out context. When any is LOW: you must compensate manually

Step 2: Engage

- Craft effective prompts at multiple levels
- This is a complicated topic
- Nobody really knows…
- But we'll give you an outline later in the lecture

Step 3: Evaluate: critically assess AI outputs against your success criteria

- The Plan-First pattern
- How do you know you're done?

The Plan-First Pattern, Part 1

- Have the AI generate a PLAN first; generate code only after plan passes review.
- A plan has a small evaluation surface (easier to review)
- Expert review: “Wait, we should use sessions not JWT for this use case”
  - plan exposes architectural issues early.
- Learner review: “I don’t know what JWT is — I should learn that before approving!”
  - plan reveals knowledge gaps.
- Cost to fix at this stage: LOW
- **If you can’t evaluate the PLAN, you can’t evaluate the CODE. Plans reveal knowledge gaps early.**

The Plan-First Pattern, Part 2: Verify that the code matches the blueprint

- After plan approval, AI generates code. Review code WITH the plan as a checklist.
- “I know what to look for because I already approved the approach.”
- **If you can’t evaluate the PLAN, you can’t evaluate the CODE. Plans reveal knowledge gaps early.**
- Advanced usage: some of this review can be automated (with a separate, isolated AI)
- But then you still have to review the review!

You also need a plan to evaluate the plan (and the code)

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

How Do You Know You're Done?

- **Whether waterfall or agile, you need to know what "done" means BEFORE you start each piece of work.**
- **Without a definition of done:**
  - "Make it better" — but better how? Better for whom?
  - Endless tweaking with no exit condition
  - "I'll know it when I see it" rarely works in practice
- **With a definition of done:**
  - Clear success criteria BEFORE you start prompting (or coding)
  - Each output can be checked against the criteria
  - The plan you approved in Plan Mode IS your definition of done
- Remember: user stories, conditions of satisfaction, etc.

If you can't evaluate the AI output, maybe you shouldn't be using so much AI

- **Easy to evaluate — use AI freely:**
  - Does the script run? Does it compile? Does the test pass? (binary yes/no — verify immediately)
- **Hard to evaluate — use AI carefully:**
  - Does the quiz have confusing questions? → Give it to 100 students and find out
  - Is this architecture scalable? → Wait 6 months in production
  - Will users find this intuitive? → Ship and measure
- **AI can help you GENERATE for any task. But for hard-to-evaluate outcomes, don’t rely on AI to EVALUATE — you need external validation: user testing, expert review, time in production.**

Steps 4-6: Calibrate/Tweak/Finalize

- **Calibrate**: Steer AI toward desired outcomes through feedback
  - Example: “That’s close, but use interfaces instead of abstract classes”
  - This is a conversation — you may calibrate 3–4 times per task. This is normal.
- **Tweak**: Manually refine AI-generated artifacts
  - Naming, edge cases, comments, style adjustments. AI output is rarely perfect — this is expected.
- **Finalize**: Document decisions and rationale
  - Create artifacts to guide future development

Steps 4 and 5: Calibration and Tweaking

- Especially applicable for non-code AI-generated artifacts
  - text
  - diagrams
  - slides
  - etc…

Step 6: Finalizing

- Before committing AI-generated code, ask yourself:
  - Can I justify the design decisions in this code to a colleague?
  - If I look at this in 6 months, will I know WHY I chose this approach?
  - Am I prepared to take responsibility if this code is wrong?
  - Would a new team member understand the design decisions?
- **If the answer to any of these is “no” — you are creating maintenance debt that future-you will pay.**
- *“It worked” is not a reason. “The AI suggested it” is not a reason. Document the actual design rationale.*

Use Design As a Way of Communicating Organization

Remember this slide from Module 08 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

Back to step 2: Engage: Craft Effective Prompts at all Levels

- We'll look at different roles that prompts can play in an AI workflow
- We'll discuss the Contract-Prompt pattern
- We'll list some myths (and non-myths) about prompting.

There are at least 4 levels of prompting with your AI

Layer	File	Scope	Who reads it
Project constraints	`CLAUDE.md`	Whole project, persistent	Auto-read every session
Architecture/rationale	`DESIGN.md`, `README.md`	Whole project, reference	Indexed and retrieved by context
Task prompts	`.claude/commands/*.md`	Per-task, reusable	Invoked explicitly via slash command
In-chat prompt	typed in session	One-off, ephemeral	Only the current session

- You can see there is some Claude-specific info here; we'll talk about other agents later

Table generated by Claude.ai

Layer 1: Project Constraint Prompts

```
## Software Stack
- Frontend: Next.js 14+ App Router, TypeScript strict
- Backend: Convex for real-time data, Supabase for auth + storage
- Auth: Clerk (never roll custom auth, we are not animals)
- Styling: Tailwind only — no CSS modules, no styled-components
## Hard Rules
- Never install a new dependency without asking first
- Never modify the database schema without showing the migration plan
- All API calls go through Convex functions, never direct Supabase
  client calls from components
- Environment variables go in .env.local, never hardcoded
  (I will find you and I will revert you)
# FAILURE CONDITIONS:
- Any component exceeds 150 lines
- Fetches data client-side when it could be server-side
- Uses any UI library besides Tailwind utility classes
- Missing loading and error states
- Missing TypeScript types on any function parameter
```

These are your guardrails

Layer 2: Architecture/Rationale Prompts

```
## 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

## Patterns
- Use server components by default, client components only when
  interactivity is required
- Error boundaries on every route segment
- Zod validation on every user input
```

- Autogenerated README.md files can be a good basis for this portion of the prompt

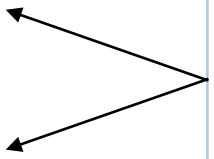
Layer 3: Task Prompts

- Carefully craft these in advance
- Put each of them in a file so they can be reused, eg

  `.claude/commands/add-subscription.md`

For Task Prompts, use the Contract-Prompt pattern

- Write prompts like like legal contracts, with four enforceable sections:
- **Goal** — the exact success metric.
- **Output Format** — the specific structure you expect.
- **Constraints** — hard boundaries that can't be crossed.
- **Failure Conditions** — what makes the output unacceptable.



Include these if the weren't covered in Layers 1 and 2

Phil Rentier, "I Stopped Vibe Coding and Started "Prompt Contracts" — Claude Code Went From Gambling to Shipping" in Medium.com, Feb. 11, 2026

Task Prompt Section 1: Goal

```
# Goal (Before: vibe) Add a subscription system to the app
- Add a subscription system to the app

# Goal (After) Implement Stripe subscription management
## Success
- a user can subscribe to one of 3 tiers (free/pro/team)
- a user can upgrade/downgrade instantly
- a user can see billing status on /settings/billings
- a free user can subscribe to Pro,
- see the charge on Stripe dashboard
- access gated features within 5 seconds.
```

The difference is testability -- each of these can be independently tested.

Task Prompt Section 2: Format

What you want delivered, and how you want it delivered

- Tell it what to hand you
- Before: `Create an API endpoint for user onboarding`
- After:

```
# FORMAT:
1. Convex function in convex/users.ts (mutation, not action)
2. Zod schema for input validation in convex/schemas/onboarding.ts
3. TypeScript types exported from convex/types/user.ts
4. Include JSDoc on the public function
5. Return { success: boolean, userId: string, error?: string }
```

Mapping the layers to other AI Agents

- Most tools have converged on at least two layers of prompts--
- Here are some tables to illustrate:

Layer 1: Project Constraints (persistent, auto-loaded)

Tool	File Location	Notes
Claude Code	`CLAUDE.md` (repo root)	Auto-read every session; supports nested files in subdirectories
GitHub Copilot	`.github/copilot-instructions.md`	Auto-attached to all chat requests; `AGENTS.md` also works and is more portable across tools
Cursor	`.cursor/rules/*.mdc`	Supports path-specific rules per subdirectory; old `.cursorrules` still works but deprecated
ChatGPT	Project custom instructions (UI only)	Set in the ChatGPT interface — not a repo file, not version-controlled

Table generated by Claude.ai

Layer 2: Task Prompts (invoked explicitly per task)

Tool	File Location	How to Invoke
Claude Code	`.claude/commands/*.md`	Type `/filename` in chat
GitHub Copilot	`.github/prompts/*.prompt.md`	Type `/promptName` in VS Code, `#promptName` in Visual Studio
Cursor	`.cursor/rules/*.mdc` (with `alwaysApply: false`)	Agent picks up relevant rules automatically, or reference with `@rulename`
ChatGPT	No repo-file equivalent	Paste manually, or upload as a file to the Project

Table generated by Claude.ai

Prompting Myths That Don’t Actually Help

- **Myth: Persona prompts improve output**
  - “You are a brilliant 10x engineer...” — Models don’t roleplay better code. Context and specificity matter, not flattery.
- **Myth: Politeness affects performance**
  - “Please” and “Thank you” — Won’t affect the model, but polite communication is a good habit for talking to humans!
- **Myth: More instructions = better**
  - 500-word system prompts often perform WORSE — key details get lost. Be concise and specific.
- **Myth: Magic phrases always work**
  - “Think step by step” — Useful for reasoning, but tools have built this in better. See: Plan Mode.
- **What actually helps: Relevant context · Specific requirements · Concrete examples · Clear success criteria**

Making sure the AI obeys the constraints

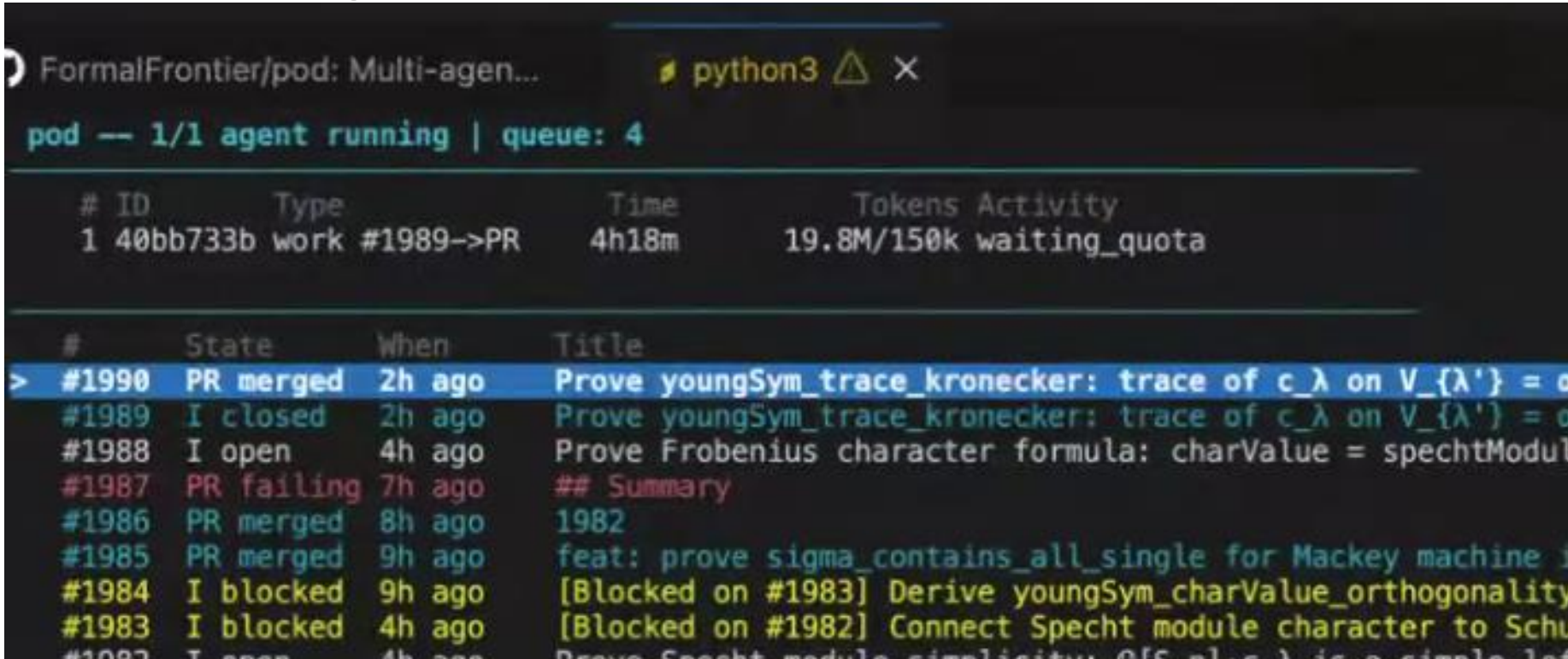
- When you start a new session, always say:

```
- Read CLAUDE.md and confirm you understand the project constraints before
doing anything.
```

- In fact, say this (or the equivalent) every so often
- Makes sure that the project constraints haven't drifted out of the context
- On startup, this asks the agent to expose ambiguities, etc., when they are easier to fix.

“Teams of agents”

- So far in 2026, there have been some unhinged, heavily hyped versions of having different agents with different instructions working together. (Search for “Gas Town Yegge” at your own risk)
- One of Prof Simmons’s colleagues at Lean FRO, Kim Morrison, recently demoed a less unhinged version of this idea, shown here.



4. A few words about AI traps

Trap #1: The “Vibe Coding” Trap

- What is “vibe coding”?
  - Evaluating only EXECUTION, not CODE. Ask AI to implement a feature. Run the app, see if it “works.” If error, describe the error to AI and repeat. Never actually read or understand the code.
- Why it leads to collapse:
  - No troubleshooting capability — you don’t understand what the code does
  - Can’t provide effective feedback — you can only describe symptoms, not problems
  - Brittle development — one change breaks everything and you don’t know why
- To effectively use AI, you must be able to EVALUATE the code itself — not just “does it run?”

Trap #2: AI Creates "Learning Debt" When Used Too Early

- **Path A — Learning First (recommended):**
  - Manual coding, mistakes, understanding errors, building mental models. Slower at first.
  - When you eventually use AI, you can evaluate and guide it effectively.
  - Solid foundation that supports long-term productivity growth.
- **Path B — AI First:**
  - Fast initial progress. But you never learn WHY the code works.
  - When complex bugs hit, you're stuck — you can't even describe the problem to AI effectively.
  - Productivity collapses and never recovers to Path A levels.

A "learning-tax" strategy can help mitigate deskilling.

- 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)

Outline

1. What is an LLM? What is an AI Coding Assistant?
   Why is context so important?
2. A 6-step Pattern for Human-AI Workflow
3. Patterns and Myths for Effective Prompting
4. A few words about AI traps

Learning Goals for this Lesson

- Now, at the end of this lesson, you should be able to
  - Explain what an AI Coding Assistant is and is not
  - Describe how to determine when using an AI assistant is or is not appropriate
  - Explain the 6-step Human-AI Workflow pattern
  - Explain the Plan-First pattern
  - Explain the Prompt-Contract pattern
  - Know how to avoid de-skilling