Vibe Coding

A Guide to Using Agentic AI in Software Development

Brendan Shea, PhD

Rochester Community and Technical College

December 4, 2025

What is Vibe Coding?

- **Vibe coding** is an AI-assisted software development technique where you describe what you want in plain English, and AI generates the code for you.
- The term was coined by Andrej Karpathy, co-founder of OpenAI, in February 2025 and quickly became widespread in the developer community.
- Collins Dictionary named "vibe coding" its Word of the Year for 2025, reflecting how rapidly this approach has entered mainstream technology culture.
- Karpathy summarizes the approach simply: "See things, say things, run things"—you observe a problem, describe it naturally, and execute what the AI produces.

Key Shift

Traditional coding requires you to learn the computer's language (Java, Python, etc.). Vibe coding inverts this: the computer learns to understand *your* language.

Why This Matters

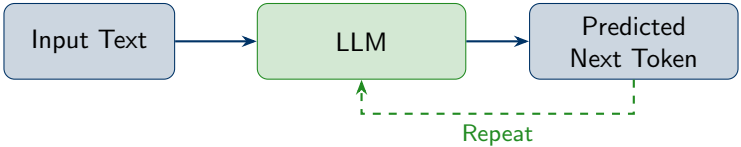
- According to GitLab's 2025 research, 78% of professional development teams now use AI-assisted coding tools as part of their regular workflow.
- This is likely how you will work in your future career—understanding these tools now gives you a significant advantage in the job market.
- Here's the key insight: knowing Java (or any programming language) makes you *better* at vibe coding, not obsolete.
- Your role shifts from "person who types code" to architect, director, and quality inspector—you guide the AI and verify its work.

Today's Goals

1. Understand how LLMs and AI agents actually work
2. Learn the process for building software with AI assistance
3. Design and prototype your own Swing application

What is a Large Language Model?

- A **Large Language Model (LLM)** is a program trained to predict "what comes next" in a sequence of text, based on patterns learned from massive datasets.
- LLMs are trained on enormous amounts of text—books, websites, documentation, and millions of lines of source code from public repositories.
- Popular examples include GPT (OpenAI), Claude (Anthropic), Gemini (Google), and Llama (Meta), each with different strengths and capabilities.
- Critical insight: LLMs don't truly "know" things or "understand" meaning—they predict statistically likely continuations based on patterns they've seen.



Exercise 1: Seeing Prediction in Action

- This exercise demonstrates how LLMs work by generating multiple completions for the same starting text.
- You'll see that each completion is plausible but different—the model is predicting likely continuations, not retrieving a "correct" answer.
- Code completions tend to be more predictable than English sentences because programming follows stricter patterns and conventions.

Try This (Using Any LLM)

1. Send: `Complete this sentence five different ways: "The programmer opened the IDE and"`
2. Notice how each completion is plausible but different.
3. Now try: `Complete this Java code five different ways: "public static void main(String[] args) {`
4. **Reflect**: Why are code completions more predictable than English sentences?

Tokens: How LLMs Read Text

- A **token** is the basic unit that LLMs process—roughly 4 characters or about three-quarters of an average English word.
- LLMs don't see individual letters or whole words; instead, they break text into these token chunks before processing.
- Code gets tokenized too: a method call like `System.out.println` might become several separate tokens.
- Understanding tokens matters because LLMs have a maximum number of tokens they can process at once—this is called the **context window**.

Original: `"Hello world"`

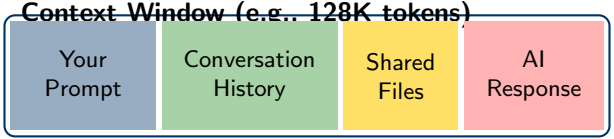
Tokenize `Hello` `_world` = 2 tokens

Code: `System.out.println`

`System` `.` `out` `.` `println` = 5 tokens

The Context Window

- The **context window** is the LLM's "working memory"—the total amount of text it can see and consider at one time during a conversation.
- Modern LLMs have large context windows from around 128K tokens (roughly 300 pages) to 1 million tokens or more.
- Everything counts toward this limit: your prompt, the conversation history, any code files you share, and the model's own responses.
- Once the context window fills up, older content gets pushed out and effectively "forgotten"—this is why breaking large projects into pieces matters.



Exercise 2: Exploring Context Window Limits

- This exercise demonstrates that LLM "memory" exists only within a single conversation—it cannot remember anything from previous sessions.
- Each new conversation starts with a completely blank slate; the model has no knowledge of who you are or what you've discussed before.
- This limitation has important implications for large coding projects: you must provide relevant context each time you start a new session.

Try This (Using Any LLM)

1. Start a **new conversation**. Ask: `What is my favorite color?`
2. The LLM doesn't know—it has no context about you.
3. Now say: `My favorite color is blue. What is my favorite color?`
4. It remembers within this conversation!
5. Start a **brand new conversation** and ask: `What is my favorite color?`
6. **Reflect**: What does this tell you about working on a project across multiple sessions?

Why LLMs Are Good at Code

- LLMs were trained on millions of public code repositories, so they have "seen" countless examples of Java programs, Swing applications, and common coding patterns.
- Code is highly structured and predictable—programming languages have strict syntax rules, and developers tend to solve similar problems in similar ways.
- Common patterns appear repeatedly across codebases: loops, conditionals, class structures, and standard library calls are all highly predictable sequences.
- Key limitation: LLMs can reproduce patterns effectively, but they don't truly "understand" program logic or verify that code actually works correctly.

What This Means for You

The LLM has seen thousands of `JFrame` setups, `ActionListener` implementations, and `ArrayList` operations. It can reproduce these patterns fluently—but *you* must verify the logic is correct for your specific needs.

Exercise 3: Testing the Limits of "Understanding"

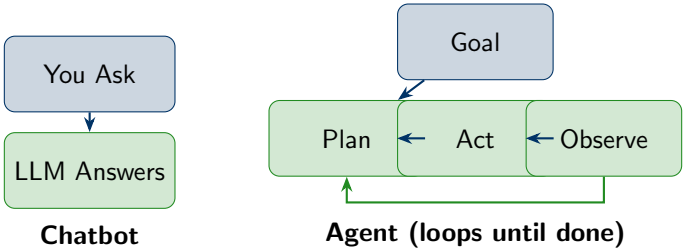
- This exercise reveals that LLMs handle common patterns well but struggle with novel or complex logic that requires genuine reasoning.
- The first task (checking if a number is even) is a pattern the LLM has seen thousands of times and will likely get correct.
- The second task requires combining date logic, calendar rules, and conditional checks—a much less common pattern that often produces buggy code.

Try This (Using Any LLM)

1. Ask: `Write a Java method that returns true if a number is even, false otherwise.`
2. It will likely get this right (very common pattern).
3. Now ask: `Write a Java method that returns true if today is the third Tuesday of the month, but NOT the 12th Tuesday of the year.`
4. **Carefully review**: Does this code actually work? Test edge cases!
5. **Reflect**: Why was the second task harder? When should you be skeptical of AI-generated code?

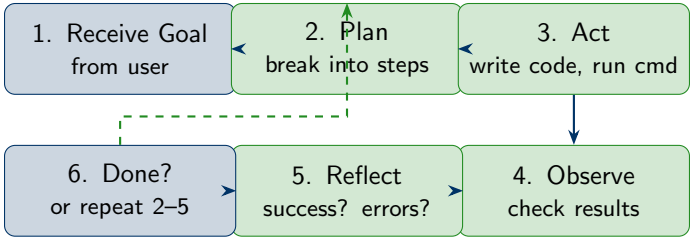
From Chatbot to Agent

- A basic LLM is like a conversation partner: you ask a question, it generates a response, and the interaction ends there.
- An **AI agent** combines an LLM with tools and a control loop, allowing it to take actions, observe results, and keep working toward a goal.
- The key difference is autonomy: agents can execute multi-step plans, recover from errors, and iterate until a task is complete.
- Analogy: asking an LLM for directions gives you a static answer; an agent is like GPS that actively reroutes when you miss a turn.



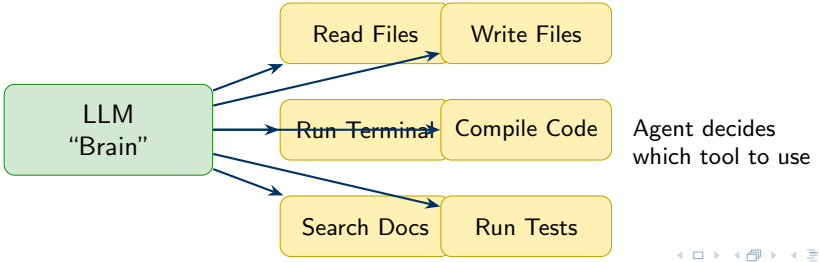
The Agent Loop

- AI agents follow a repeated cycle: receive a goal, make a plan, take action, observe the results, and decide what to do next.
- This loop continues until the agent determines the goal is achieved, encounters an unrecoverable error, or you manually stop it.
- The "observe" step is critical: agents can detect compilation errors, test failures, or unexpected output and attempt to fix problems automatically.
- This is what makes agent-based coding powerful—but also why human oversight remains essential.



Tools: Giving the LLM Hands

- On their own, LLMs can only generate text—they cannot actually execute code, create files, or interact with your computer in any way.
- **Tools** are capabilities that let an agent take real actions: reading files, writing code, running terminal commands, compiling programs, and more.
- The agent's LLM decides *which* tool to use and *when* based on the current goal and what it observes after each action.
- This is what transforms a passive text generator into an active coding assistant that can actually build and test software.



Exercise 4: Thinking Like an Agent

- This exercise helps you understand how agents break down tasks into steps and anticipate what could go wrong at each stage.
- By asking the LLM to *plan* rather than *code*, you learn to think at the architectural level—a crucial skill for directing AI effectively.
- Understanding the agent's decision-making process helps you write better prompts and catch problems before they compound.

Try This (Using Any LLM)

1. Ask: `If you were an AI agent with tools to read files, write files, and run terminal commands, describe step-by-step how you would add a "Save" button to an existing Java Swing app. Don't write code---just describe your plan.`
2. Read the plan carefully. Does it make sense?
3. Now ask: `What could go wrong at each step? How would you detect and fix each problem?`
4. **Reflect**: How is asking for a *plan* different from asking the agent to "just do it"?

Why Agents Still Need You

- Agents optimize for "looks complete" rather than "actually correct"—they may produce code that compiles but contains subtle logic errors.
- The agent only knows what you tell it; it cannot read your mind about unstated requirements, edge cases, or how the software should "feel" to users.
- Agents can get stuck in loops, make things worse while trying to fix errors, or confidently produce insecure or inefficient code.
- Your role is essential: you are the **architect** who designs the system, the **director** who guides the work, and the **inspector** who verifies quality.

The Human-Agent Partnership

Agent's Job:	Generate code, run commands, iterate on errors
Your Job:	Define requirements, design architecture, verify correctness

Setup & Tools Overview

- To use AI coding agents, you need an account with an AI provider (like GitHub Copilot, which is free for students) and an editor that supports agent mode.
- The specific tools and interfaces change rapidly—detailed setup instructions are on the course website and will be updated as needed.
- What matters most is understanding the *concepts*: once you understand how agents work, you can adapt to any specific tool.

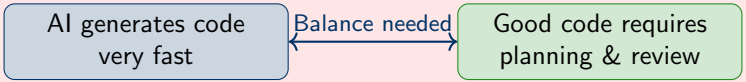
Key Components (Tools May Vary)

1. **AI Provider Account**: GitHub Copilot (free via Student Developer Pack), Claude, ChatGPT, or others
2. **Code Editor**: VS Code, IntelliJ, or other IDE with AI integration
3. **Agent Mode**: The autonomous mode where AI can read/write files and run commands
4. **Chat Mode**: For asking questions, getting explanations, and requesting snippets

The Vibe Coding Paradox

- AI makes generating code dramatically faster—what once took hours can now take minutes, and what took days can take hours.
- However, this speed also makes generating *bad* code faster; without discipline, you can create a tangled mess in record time.
- The **vibe coding paradox**: the easier it becomes to generate code, the more important process and planning become.
- Speed without direction means you get lost faster—you need a clear destination before you start moving quickly.

The Core Tension



Exercise 5: The "Everything at Once" Disaster

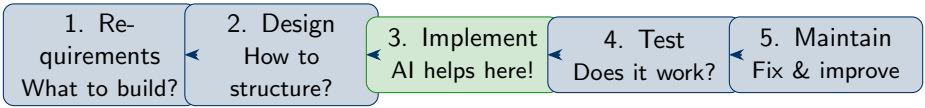
- This exercise demonstrates why asking for too much at once leads to problematic results—even when the AI appears to succeed.
- You'll experience firsthand how difficult it is to understand, debug, and modify code when you didn't build it piece by piece.
- The goal is not to discourage ambition, but to show why breaking work into smaller pieces produces better outcomes.

Try This (Using Any LLM)

1. Ask: `Write a complete Java Swing task manager with categories, due dates, priorities, search functionality, and the ability to save/load from a file.`
2. Examine the result and answer these questions:
   - How many lines of code did it generate?
   - Can you explain what every section does?
   - Does it actually compile and run correctly?
   - If there's a bug, where would you start looking?
3. **Reflect**: What would have been a smarter *first* prompt?

The Software Development Lifecycle (SDLC)

- The **Software Development Lifecycle (SDLC)** is a structured process that professional developers follow to build reliable, maintainable software.
- AI accelerates the Implementation phase dramatically, but it cannot replace the other phases—in fact, they become more important.
- Skipping Requirements or Design means the AI has no clear target; skipping Testing means bugs slip through; skipping Maintenance planning means future changes become painful.



Incremental Development: The Golden Rule

- **Incremental development** means building software in small, testable pieces rather than trying to create everything at once.
- Each piece should be small enough to understand completely, test thoroughly, and fix easily if something goes wrong.
- This approach aligns perfectly with how AI agents work best: focused tasks with clear success criteria produce much better results than vague, sprawling requests.
- Professional developers call this "Agile" methodology—and it's even more important when working with AI.

The Incremental Process

1. Build the smallest working piece first (e.g., just a window)
2. Test it—does it work correctly?
3. Add the next piece (e.g., a list display)
4. Test again—does everything still work?
5. Repeat until the application is complete

Exercise 6: Breaking Down a Project

- This exercise uses the LLM as a planning partner to help you decompose a project into manageable, buildable pieces.
- The key skill here is learning to think architecturally: what are the components, and in what order should they be built?
- Save the output from this exercise—you will use this breakdown to guide your actual implementation later.

Try This (Using Any LLM)

1. Choose a simple app idea (or use: "a flashcard study app")
2. Ask: `I want to build a [your app] using Java Swing. Help me break this into 5--6 small pieces I could build one at a time. Each piece should be testable on its own. List them in build order.`
3. Review: Does the order make sense? Would each piece work alone?
4. Ask: `For piece #1, what would I see on screen when it's working?`
5. **Save this list**—you'll use it for your project!

Anatomy of an Effective Prompt

- A **prompt** is the instruction or request you give to an LLM; the quality of your prompt directly determines the quality of the output.
- Effective prompts share four key elements: context about what exists, a specific task, constraints to follow, and guidance on quality.
- Think of prompting as giving instructions to a skilled but literal-minded assistant who has no background knowledge about your specific project.

The Four Elements of a Good Prompt

Element	What to Include
1. Context	What already exists? What are you building?
2. Specific Task	What exactly should this step accomplish?
3. Constraints	Language, libraries, patterns, limitations
4. Quality Guidance	"Add comments," "Keep it simple," "Explain first"

Exercise 7: Good Prompt vs. Bad Prompt

- This exercise lets you directly compare the results of a vague prompt versus a detailed, well-structured prompt.
- You'll see how specificity dramatically improves the usefulness and correctness of AI-generated code.
- The extra time spent writing a good prompt is recovered many times over by reducing debugging and confusion.

Try Both Prompts (Using Any LLM)

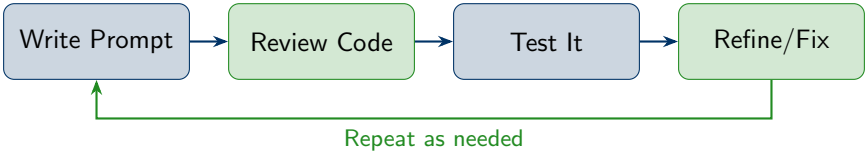
Prompt A (vague): `Make a Java GUI with a list and buttons.`

Prompt B (specific): `Create a Java Swing JFrame (500x400 pixels) with BorderLayout. In the center, add a JList displaying String items. At the bottom (SOUTH), add a JPanel with two buttons labeled "Add" and "Remove". No functionality yet---just the layout. Add comments explaining each component.`

Compare the outputs: Which is more useful? More understandable? Try running both—which works better? **Reflect**: What made Prompt B more effective?

The Iteration Mindset

- **Iteration** means repeatedly refining your work through cycles of attempt, feedback, and improvement—this is normal and expected in vibe coding.
- Don't expect perfection on the first try; even expert prompters go back and forth with the AI to get the result they want.
- When something doesn't work, describe *specifically* what went wrong rather than just saying "it doesn't work" or "try again."
- Treat the interaction like a conversation with a colleague: you guide, it types, you review, you refine together.



Exercise 8: The Iteration Loop in Practice

- This exercise gives you hands-on experience with the iteration loop by building on code from Exercise 7 and adding functionality.
- You'll practice the most important vibe coding skill: describing problems specifically when something doesn't work.
- Notice how many back-and-forth exchanges it takes—this is completely normal, not a sign of failure.

Try This (Continuing from Exercise 7, Prompt B)

1. Take the GUI code from Prompt B (the JFrame with JList and buttons)
2. Ask: `Add an ActionListener to the "Add" button. When clicked, show a JOptionPane input dialog asking for text, then add that text to the JList.`
3. Test it—does it work? If not, describe the specific problem.
4. If it works, ask: `Now make the "Remove" button delete the currently selected item from the list.`

Choosing Your Project

- For the remainder of this lecture and your homework, you will design and build your own Java Swing application using AI assistance.
- Choose something simple that interests you—enthusiasm helps you push through challenges, and simplicity ensures you can actually finish.
- Your app must have a graphical interface with at least three interactive components (buttons, text fields, lists, etc.).
- Remember: you can always add more features later; start with a minimal version that actually works.

Project Ideas (Pick One or Invent Your Own)

Task manager / To-do list	Flashcard study app
Simple quiz game	Habit tracker
Recipe organizer	Tip calculator
Unit converter	Countdown timer
Personal diary / journal	Password generator

Exercise 9: AI-Assisted Requirements

- **Requirements** define what your software should do—they are the foundation that everything else builds upon.
- In this exercise, you'll use the LLM as a collaborator to help you think through what features your app needs.
- The goal is not to accept everything the AI suggests, but to use its suggestions as a starting point for your own decisions.

Try This (Using Any LLM)

1. Decide on your app idea from the previous slide
2. Ask: `I'm a beginner building a [your app] in Java Swing. Help me define clear requirements. What are 4--5 core features a simple version should have? Keep it realistic for a beginner.`
3. Review the suggestions—are they too ambitious? Too simple?
4. Ask: `For each feature, what would the user actually DO? (click, type, select, etc.)`
5. **Write down your final list of 4–5 features**

Exercise 10: AI-Assisted Design

- **Design** determines how your software will be structured—what components exist and how they fit together.
- This exercise helps you visualize your application's interface and plan the order in which you'll build each piece.
- Creating a sketch (even a rough one on paper) before coding dramatically reduces confusion and wasted effort.

Try This (Using Any LLM)

1. Describe your app and features from Exercise 9 to the LLM
2. Ask: `Describe what the main window should look like. What Swing components would I need? Where should each go?`
3. Ask: `What Java classes or data structures will I need to store the app's data?`
4. Ask: `Break this into 5--6 pieces I should build incrementally. What order?`
5. **Sketch the window on paper and number your build order**

Exercise 11: Start Building Your Project

- Now it's time to start actually building—using everything you've learned about incremental development and effective prompting.
- Begin with piece #1 from your design (usually the basic window with layout) and don't move on until it works correctly.
- Use the prompt pattern: provide context, specify the task clearly, add constraints, and request comments for learning.

Start Building (Using Agent Mode or Chat)

1. Create a new project folder for your application
2. Build **only piece #1** from your plan (e.g., basic window with layout)
3. Suggested prompt pattern: `Create a Java Swing [component] for a [your app]. [Specific details about size, layout, labels]. Add comments explaining each part. This is step 1---keep it simple.`
4. **Test it**: Does the window appear? Does it look right?
5. If yes: save your work and move to piece #2
6. If no: describe the problem specifically and iterate

Vibe Coding Best Practices

- These seven practices separate successful vibe coders from those who end up frustrated with tangled, broken code.
- Notice that most of these practices happen *before* or *after* the AI generates code—the human work remains essential.
- Treat these as habits to build; they will serve you throughout your programming career, with or without AI assistance.

Seven Habits of Effective Vibe Coders

1. **Plan before prompting**—know what you're building
2. **Build incrementally**—one working piece at a time
3. **Review everything**—never accept code you don't understand
4. **Test after each change**—catch problems early
5. **Ask for explanations**—use AI to learn, not just produce
6. **Keep prompts focused**—one task per prompt works best
7. **Iterate willingly**—back-and-forth is normal, not failure

Common Pitfalls to Avoid

- Learning from others' mistakes is faster than making them all yourself—these are the most common ways vibe coding goes wrong.
- Most pitfalls stem from either asking for too much at once or accepting results without sufficient scrutiny.
- When you catch yourself falling into one of these traps, pause and return to the fundamentals: plan, build small, test, review.

Six Ways Vibe Coding Goes Wrong

1. Dumping huge requirements into a single prompt
2. Accepting code you don't understand
3. Skipping testing because "the AI wrote it"
4. Not learning fundamentals ("I'll just ask AI")
5. Ignoring security, edge cases, and error handling
6. Giving up when the first result isn't perfect

Key Takeaways & Homework

What You Learned Today

- **LLMs** predict text based on patterns; **agents** add tools and loops to accomplish goals autonomously
- Process matters more than ever: requirements → design → incremental build → test
- Good prompts are specific, contextual, and focused on one piece at a time
- Your Java knowledge makes you *better* at vibe coding, not obsolete—you are the architect

Homework Assignment

1. Complete your Swing application (minimum 3 working features)
2. Submit the following:
   - Your requirements list (from Exercise 9)
   - Your design sketch and build order (from Exercise 10)
   - Three example prompts you used and what they produced
   - Your working Java code
3. Be prepared to demo your app and explain how you built it