

10

Vibe coding with Cursor

This chapter covers

- Using vibe coding to program through conversation
- Cursor's AI development environment
- Building a retro arcade-style game using natural language prompts
- Mastering context management and prompt engineering
- Using AI agents for quick prototyping

You already know modern AI tools are revolutionizing the way software is developed. The tasks they help with range from simple code completion to conversational programming. At one end of the spectrum lies *vibe coding*. This approach uses natural language, letting developers guide AI agents with easy prompts instead of complex specs. Tools such as Cursor and Windsurf lead this trend. They offer immersive environments where AI can explore codebases, make changes across files, and run full workflows. You don't need to be a programmer but only possess basic tech skills.

For software developers, these tools open doors for quick prototyping. They allow learning new technologies and creating projects where speed matters more than detailed documentation. While vibe coding shouldn't replace structured practices for important applications, it adds value to traditional workflows. It helps you quickly validate ideas, explore new frameworks, and turn concepts into working prototypes faster than ever.

Now let's build an application using vibe-coding methods and Cursor, a popular vibe coding tool. You'll come to understand how this approach differs from other forms of AI-driven assistance we have explored before. You'll also learn best practices for getting the most out of Cursor and vibe coding.

10.1 *What is vibe coding?*

This term is new enough that folks are still arguing about what it means. I'll give it my best shot at a definition. Vibe coding is a programming style where users create software by means of simple language prompts. They interact with an LLM for coding, like we've been doing throughout this whole book. The AI generates the code, shifting the human role from manual coder to guide and tester. Users often accept and use this code without fully grasping its details. They trust the AI's suggestions and go with the flow.

Andrej Karpathy introduced the term in February 2025 (<https://x.com/karpathy/status/1886192184808149383>). He explained that it's "not really coding—I just see things, say things, run things, and copy-paste things, and it mostly works." This method is very conversational and can even use voice commands. Vibe coding aims to simplify software creation for those without traditional programming skills. In this book, we've done the opposite, planning and understanding every line of generated code and fitting it in with the code we write ourselves. Does that mean vibe coding is bad and should be avoided?

As defined previously, vibe coding has its place in the programming world. It helps people who aren't programmers turn their ideas into a product. Fun things such as games and 3D demos are created and released on the internet by people who don't understand any of the code. And they turned an idea into a product that may have otherwise never seen the light of day. That's a positive thing. It also removes some of the barriers to becoming a developer, and if people get a taste of making stuff and build some confidence, I'm all for it.

How can a programmer utilize vibe coding? It's counterintuitive for us to let a machine do all the work. Most programmers I know love the problem solving, algorithms, and control of coding things by hand. But there are some ways we can use it as well:

- *Rapid prototyping*—If you want to build something fast for a proof of concept, vibe coding is your friend. You know what you want and what it should look like. You can easily spend a weekend letting the LLM generate everything for you and build something that might otherwise take you weeks. As a developer, you have a distinct

advantage. You know what to ask for. You can build something fast and get in front of stakeholders, then go back and build “the real thing” your way later.

- *Unfamiliar technologies*—If you’re not familiar with a new language or library, you can vibe code something together and see how it works. If you like what you see, you can dive in and learn more. I recently tried this with `Three.js`. I don’t know much about the library or how to use it. I let Cursor take control and build a prototype of a 3D game. Once I saw what it was capable of, I went back and learned more about how it works.
- *Building stuff that doesn’t matter*—You would never want to board a plane where the developers vibe coded the flight control software. Let’s be honest, we aren’t writing aircraft software every day. That simple internal tool to solve a quick problem? A fun video game for you and your friends to play? These don’t need extensive tests and documentation. They don’t need to be audited and built with the most reliable and performant code ever written. You can just let the LLM build it. You can focus on creativity and experimentation over technical details.

There is an important caveat here. Many programmers and computer scientists are sounding the alarm about security concerns with vibe-coded software. And the concerns are valid. If you’re working on production software in an enterprise, or software that deals with sensitive data of any type, do not vibe code this and release it to the world. Remember an LLM cannot be held accountable, but you can. Use caution.

10.2 What is Cursor, and why is it different?

We will work with Cursor, an extremely popular tool, often used for vibe coding. This is because of the immersive and agentic experience this tool provides. There are many products in this field, including Windsurf, Lovable, Bolt, and others. While they’re very popular with vibe coders, they aren’t *vibe-coding tools*. They’re professional-level tools used for structured and supervised code generation as well. So, what’s the difference between these and the tools we’ve been working with?

10.2.1 The interface

One of the primary differences between Cursor and the other tools is the interface. We’ve used tools that are plugins for Visual Studio Code and other popular IDEs. Cursor (and the other tools I mentioned) have their own interface. They’re forked from VS code in many cases. This approach helps the user feel more immersed in the product. The entire IDE is built around the LLM and its usage.

The interface is only a small piece. Cursor, along with tools such as Windsurf and Lovable, stands out from traditional AI coding assistants such as GitHub Copilot. It offers deep integration of AI agents and autonomous workflows.

Cursor’s main feature is agent mode. In this mode, the AI can explore your codebase on its own. It reads documentation, edits multiple files, and even runs terminal commands. This allows it to handle complex tasks from start to finish. You stay informed for supervision and approval. For example, if you ask Cursor to “add user authentication,”

it will search your project, plan the changes, implement them, run tests, and summarize its actions. You gain more than code suggestions. You get an AI codeveloper that can reason and execute multistep tasks.

10.2.2 *Project-wide context and customization*

Unlike traditional IDE plugins that focus on a single file or need manual context, Cursor's AI analyzes your entire project. It understands structure, dependencies, and coding patterns. This enables features such as

- Multifile code generation and refactoring
- Automated error detection and linting across the whole codebase
- Context-aware chat, where the AI can answer questions about any part of your project or documentation

Additionally, Cursor supports project-level customization through files such as `.cursor-rules` (<https://docs.cursor.com/context/rules>), allowing teams to embed their coding standards and best practices directly into the AI's behavior. This feature ensures that generated code consistently follows your team's conventions, reducing review time and improving quality.

Cursor provides a type of experience where you (theoretically) can build applications simply by entering a series of prompts until the application is complete. In my experience, that's true, with smaller projects. Once the application gets large enough, it gets you about 80% of the way there, and you still need to edit things manually to make them work. Let's dive into using Cursor and do a little vibe coding.

10.3 *First concept*

In many demos, you'll see someone open an IDE and start coding without any plan. While I understand that planning isn't the point, we should do a few things that will save us some time later down the road. Since we're developers and know what to ask for, we can be more specific with requests. We know part of essential prompt engineering is being specific. So, I'll outline what I'm looking for. I'm thinking of building a fun "Frogger" clone in Python.

I want to build

- 1 A simple game where you attempt to get from the bottom of the screen to the top.
- 2 You need to cross a river that blocks your path.
- 3 The river has logs that travel horizontally, and it is possible to cross the river by jumping on and off the floating logs.
- 4 I want it written in Python, using Pygame.
- 5 The environment is a set of lanes where the goal is to get from the bottom of the screen to the top. The logs should be spaced appropriately so that it is possible to jump log to log and cross the river.
- 6 If you jump in the river (no log), you have to start over.

This is a clear enough outline to get started. Now I can begin giving it some thought to craft a detailed prompt for Cursor. You can even use Cursor or another LLM to refine your prompt. We want this initial prompt to be as thorough as possible.

I have created a simple project named “Pyfrog.” The first step will be to send a detailed, concrete prompt to get things started. We begin by reviewing the initial prompt and identifying the key elements we’d like to incorporate for a strong opening.

10.4 The initial prompt to build our game

In my experience with vibe coding, the more effort you put into your initial prompt, the better. You will iterate again and make many changes if you don’t get clear from the start. You need to think deeply and try to cover all the details of what you’re looking for up front. Otherwise, Cursor (or other tools) will take off and build a NodeJS application or something that you didn’t intend. Here is the prompt I will use to start our game:

JM

You are an expert Python game-dev assistant. Generate full, runnable code for a minimalist Frogger-style game in one file that a user can copy-paste and run immediately. Goal: Cross the river. move a player sprite from the bottom safe bank to the top safe bank without falling in.

Constraints & Specs

Tech stack: Python 3.10+, Pygame 2.x. No external assets (draw shapes with `pygame.draw`).

Window: 800 × 600 px, caption “River Run.” Use a fixed FPS = 60.

Lanes:

Total rows = 6

Row 0 (top) – safe goal bank (grass green)

Rows 1-4 – river lanes containing logs

Row 5 (bottom) – starting bank (grass green)

Logs

Each river lane spawns rectangular logs of random length (80-160 px) and speed (60-140 px/s).

Direction alternates per lane.

When a log fully exits the screen, respawn it just off-screen on the opposite side (wrap).

Player

40 × 40 px blue square.

Controls: WASD → move one full row/column at a time (grid hop), snapping to lane centers.

If player’s rect overlaps any log rect in the current river lane, treat as “standing on log” and move horizontally with that log’s velocity each frame.

If player is in a river lane and not on a log → fall = reset to start bank (score unchanged).

Scoring & Win Condition

score += 1 each time the player successfully reaches the top safe bank.

Upon success, display a centered semi-transparent overlay: “Congratulations! Score: (Press R to play again)” and pause the game until R is pressed.

Code Quality

Follow PEP 8 (snake_case names, ≤79 character lines).

Use `dataclass` where convenient (e.g., `Log`).

Include type hints and brief docstrings for all functions/classes.

Organize logic into small functions: `handle_input`, `update_game_state`, `render`, etc.

This prompt should give Cursor an excellent starting point for our application. By being specific, you can cut down the number of times you need to run prompts. One thing that’s also different about Cursor and similar tools is that, if you use premium models, extra prompts will cost you money. You may want to cut down on the number of prompts. Before we use this prompt, let’s familiarize ourselves with the Cursor basics.

10.5 *Cursor basics*

Cursor is a fork of Visual Studio Code, so the interface is largely similar. There are some key differences, though, that we’ll cover here. You may notice that when you load up Cursor and turn on the light theme, it looks nearly identical (figure 10.1).

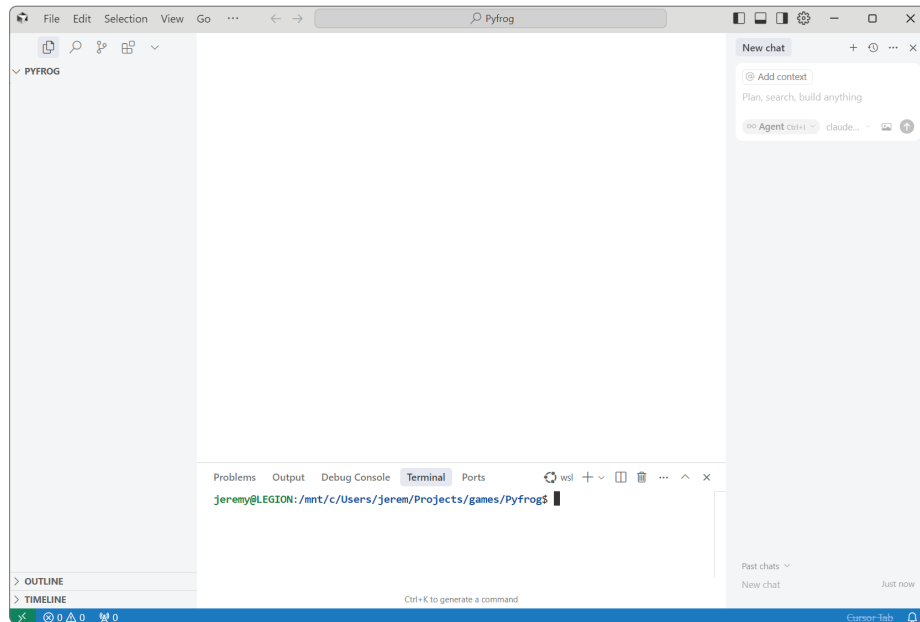


Figure 10.1 Cursor looks nearly identical to Visual Studio Code, especially when you don’t use Cursor’s special theme.

You may notice a chat window along the side, which is similar to other interfaces we’ve used. However, Cursor is more immersive, as you’ll learn soon. I have dropped the prompt from above directly into a new chat, and it’s given me a response. Figure 10.2 shows what the window looks like now.

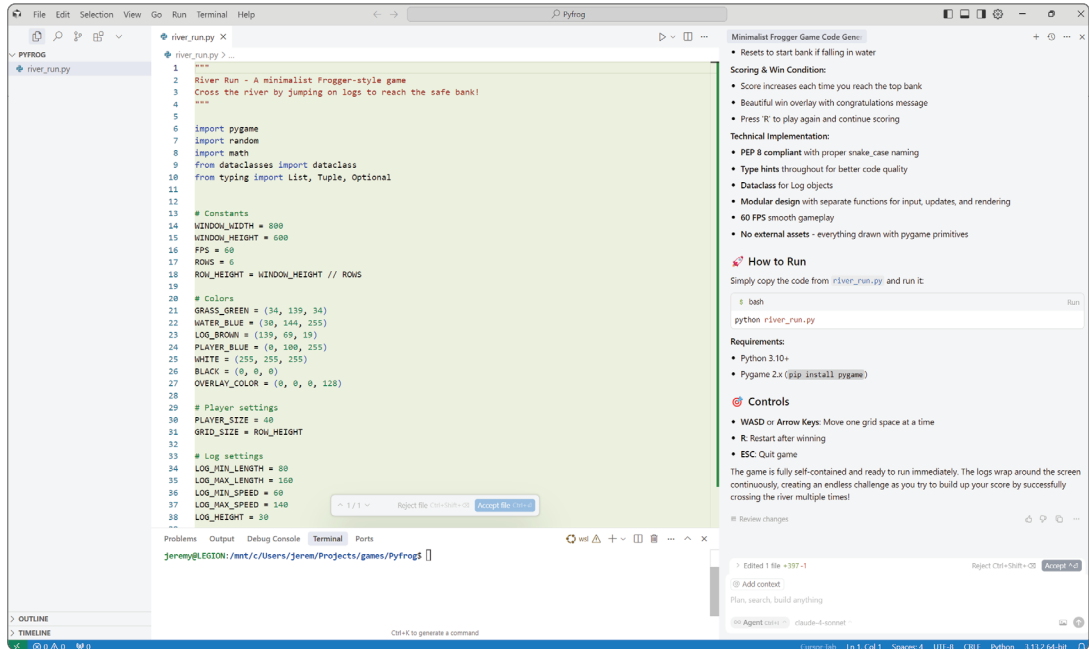


Figure 10.2 When you add a prompt into the chat window, it will come up with a plan, some actions, and then do a diff view on your code, like many tools we’ve worked with.

10.5.1 Giving feedback

One thing you’ll notice is a feedback mechanism. There are a few changes at the bottom of the result:

- Review changes
- Thumbs up
- Thumbs down
- Copy

These are self-explanatory. You can review the proposed changes and give feedback on the suggestion. Whether you want to give feedback is a personal choice. I give feedback as much as possible to help them improve the product. The last button copies the result, usually in Markdown format.

Next, you'll see the files that were changed. In my case, it was `river_run.py`. It was the only file created, so it was the only one modified (figure 10.3).



Figure 10.3 Chat window immediately after a result. You can give feedback, review changes, and see what file was changed.

10.5.2 Adding context

This is your key to one of the most powerful things about Cursor—adding context. You can tell Cursor what context to focus on. The menu is shown in figure 10.4. It includes

- *Files and folders*—Your application code or imported libraries.
- *Code*—Selected code within your application.
- *Docs*—You can use Cursor to search through documentation.
- *Git*—Use a git repository as context.
- *Past chats*—You can use your past chats with Cursor as context.
- *Cursor rules*—These are a set of rules you can set for Cursor, and they can be used as context as well.
- *Terminals*—Use your terminal history and outputs as context to answer your questions.
- *Lint errors*—If you have a linter setup, it can use those errors as context when problem solving.
- *Web*—You have Cursor search the web for your answer.

This set of options and fine-tuning context is one of the things that makes Cursor great. When you're vibe coding an application, you'll have many conversations with the LLM. Being able to tune this is crucial.

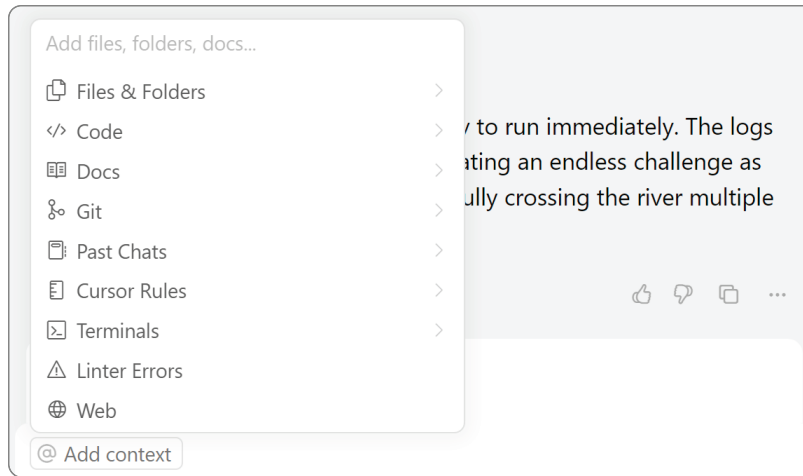


Figure 10.4 Different ways to add context in Cursor, offering a range of choices to help you refine your search or decide what to include in your questions

Why not just include the entire app as context?

You may be wondering why we must add individual files to our context. Why not include the whole application? Doing so can be a disadvantage. Loading your entire application can introduce a lot of noise and irrelevant information. It can dilute the AI's focus, leading to slower performance and less accurate suggestions. Large contexts can also hit token limits on popular models fast, costing you money.

By selectively choosing individual files, you provide a more focused, intentional context that helps the LLM work more efficiently and accurately.

10.5.3 Selecting a mode

There are three main modes for Cursor: Agent, Ask, and Manual. As shown in figure 10.5, you can switch between them with ease from the chat window. Here are how these modes work:

- *Agent*—Here you can plan, search, build, and give instructions. Crucial to vibe coding, you can say something like, “create a database interface for this application.” The agent will attempt to complete exactly what it is asked.
- *Ask*—This mode is great for asking questions. You can ask something like, “How does this API work?” Or, you can use more detailed questions such as, “Is there a better way to do file handling here?” It's great for general questions.
- *Manual*—This mode is for asking questions directly to the LLM, without using the tool. This is great for general-purpose prompts, but I have found little use for them.

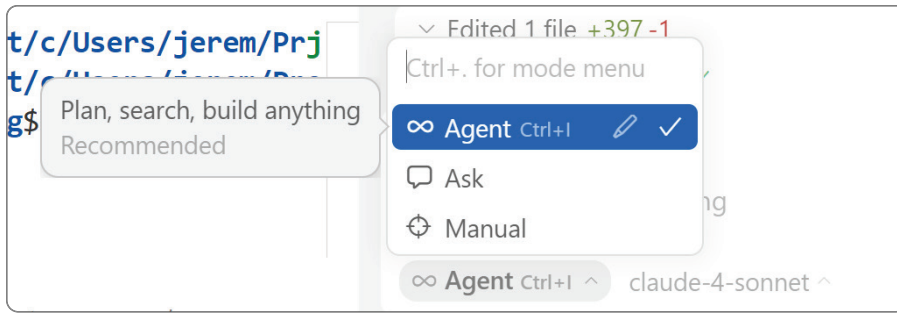


Figure 10.5 Switching modes in Cursor is handy and easy, depending on what you need from the tool. I use the Agent mode the most when vibe coding.

10.5.4 Model selection

Selecting the right model can be crucial to your success while coding. What's the best model? That's entirely too subjective, and even if I have an answer, by the time you read this, it will be something different. The big AI companies are competing for the best models and making great strides. Focusing on code generation seems to be a priority for all of them. The big takeaway here is you can choose which model to work with for each prompt (figure 10.6). Keep in mind some of these models may cost you additional money on top of your current Cursor account. You can also add custom models if you like.

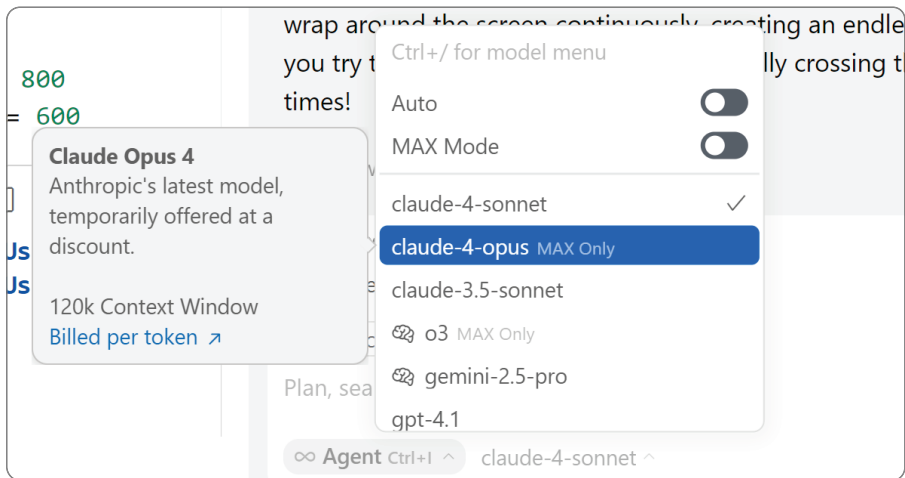


Figure 10.6 You can select different models to work with. This changes frequently, but Cursor will let you know when you'll be charged for using certain models. Generally, the newest models, or thinking models, come at a premium.

10.5.5 MAX mode

Here is a special note about this switch. If you run in this mode, it maxes out your context windows, meaning you can have very large conversations and include a bunch of files in your context. However, this can greatly affect your billing. I've talked to folks who have spent hundreds of dollars this way but had big problems to solve. To be on the safe side, it's best not to enable it if you're cost-conscious. As shown in figure 10.7, you can enable or disable it at any time and view pricing models.

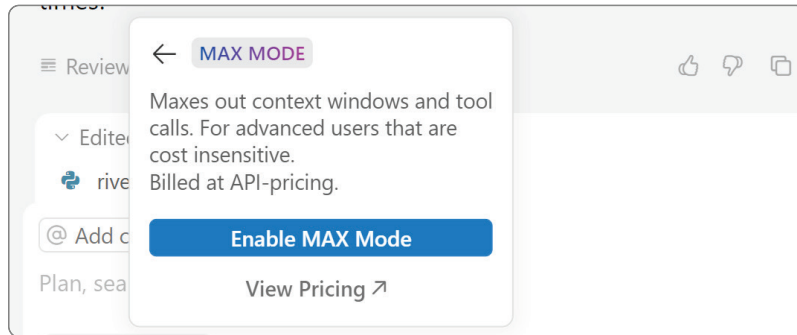


Figure 10.7 In the chat window, you can enable or disable MAX mode at any time, and view pricing for it.

Now that you have a good idea for how Cursor works, let's return to our game that we are vibe coding into existence.

10.6 Results from the first prompt

Earlier, we created a detailed prompt to start out with for this application. We want to be clear as possible and try to get an accurate start. Most importantly, we wanted to dictate the language and framework used for the application. This way, I don't have to delete it and start over. There may be cases where you don't even want to specify that much and let the tool choose. Let's look at the output our prompt created:



I'll create a complete Frogger-style game that meets all your specifications. Let me generate the full, runnable code in one file.

Perfect! I've created a complete, runnable Frogger-style game called "River Run" that meets all your specifications. Here's what the game includes:

🎮 Game Features

Core Gameplay:

6-row layout: Top/bottom safe banks (green) + 4 river lanes (blue)

Grid-based movement: WASD or arrow keys for precise lane hopping

Log mechanics: Random-sized logs (80-160px) moving at different speeds (60-140 px/s)

Alternating directions: Each river lane moves logs in opposite directions

Collision system: Stand on logs to ride them, fall in water if you miss

Player Mechanics:

40×40px blue square player

Snaps to lane centers when moving

Moves with log velocity when standing on one

Resets to start bank if falling in water

Scoring & Win Condition:

Score increases each time you reach the top bank

Beautiful win overlay with congratulations message

Press 'R' to play again and continue scoring

Technical Implementation:

PEP 8 compliant with proper `snake_case` naming

Type hints throughout for better code quality

Dataclass for Log objects

Modular design with separate functions for input, updates, and rendering

60 FPS smooth gameplay

No external assets - everything drawn with pygame primitives

🔧 How to Run

Simply copy the code from `river_run.py` and run it:

```
python river_run.py
```

Requirements:

Python 3.10+

Pygame 2.x (pip install pygame)

🕹️ Controls

WASD or **Arrow Keys:** Move one grid space at a time

R: Restart after winning

ESC: Quit game

The game is fully self-contained and ready to run immediately. The logs wrap around the screen continuously, creating an endless challenge as you try to build up your score by successfully crossing the river multiple times!

One of the things that surprised me is that the tool attempted to build a runnable game from one prompt. This is what vibe coders look for, something quick to run. There will be iterations of course, but let's see how this application runs.

10.7 *Running our game for the first time*

Amazingly, this game ran on the first try. We created an actual game with a single prompt. This is a big complement to the tool, and it underscores the importance of

adding details to your prompts when vibe coding. Figure 10.8 shows the screen that pops up when you run the application.

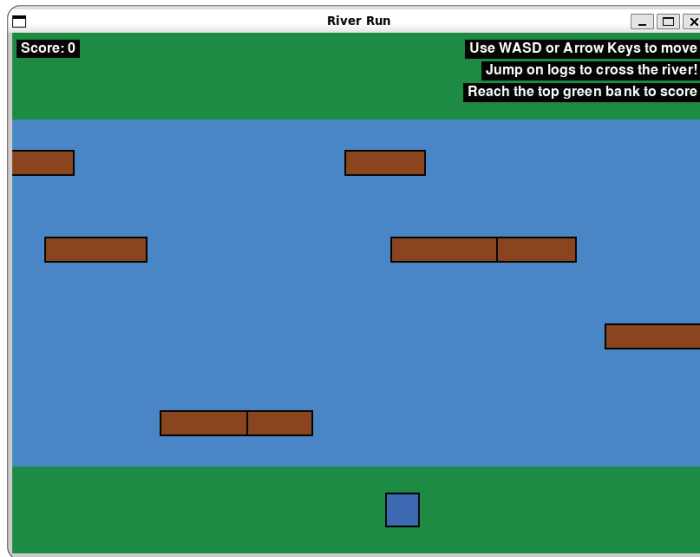
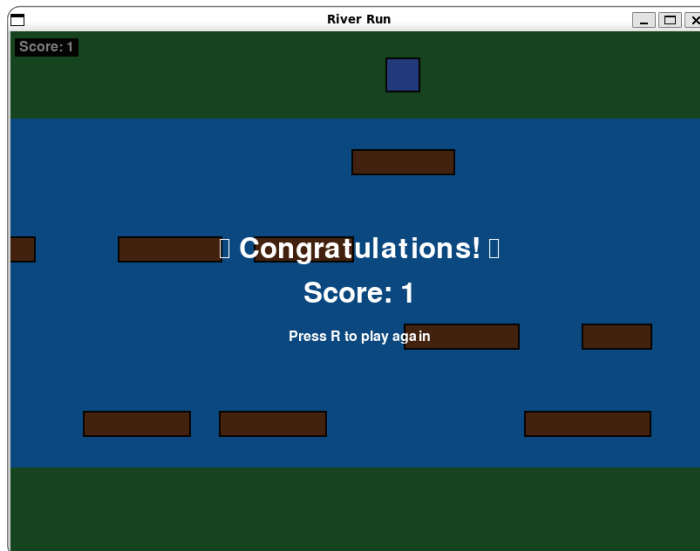


Figure 10.8 The loading screen of our application. This is the first screen you see when it loads up. It includes directions for play.

The game looks just like I'd imagined. It's playable as well, as I soon figured out how to cross the river to the other side, as shown in figure 10.9, the second screen of our game.




10.9 The second and final screen of the game that shows when you cross the river successfully

Our game is working great and was created from a single prompt. However, I'd like to make some changes. Let's see how that works.

10.8 Making changes to our game

Something you may have noticed in figure 10.8, the logs seem to overlap. This happens frequently, and it makes gameplay a little confusing, as the logs merge into each other. I would like to fix this by examining the code. However, in the spirit of vibe coding, I will let the tool fix it. I will also use plain English prompts for this, as that is a part of the methodology here. We can pretend we aren't programmers; we're just someone building a game.

Here is the prompt I will use:

 The logs move at a good speed and it's possible to cross the river. Some of the logs run into each other and overlap. Please fix it so they are spaced apart, move at the same speed, and don't overlap.

Let's see how Cursor responds to this request.



I'll fix the log spacing and movement issues to ensure they don't overlap and move at consistent speeds within each lane.

Perfect! I've fixed the log spacing and movement issues. Here are the key improvements:

Fixed Issues:

- 1 **Consistent Speed per Lane:** All logs in the same lane now move at the same speed, preventing faster logs from catching up to slower ones.
- 2 **Proper Spacing:** Added a `LOG_SPACING` constant (200px) that ensures logs maintain a minimum distance between them.
- 3 **Smart Respawning:** When a log goes off-screen, the new log spawns with proper spacing relative to existing logs in that lane, maintaining the gap.
- 4 **Fixed Number of Logs:** Each lane now consistently has 3 logs instead of a random 2-3, making the spacing more predictable.

How It Works:

- **Initial Spawn:** Logs are positioned with calculated intervals across the screen width plus extra spacing for smooth wrapping
- **Speed Consistency:** One speed is chosen per lane and applied to all logs in that lane
- **Respawn Logic:** New logs spawn at the correct distance from existing logs to maintain the spacing pattern
- **Direction Handling:** Left-moving and right-moving lanes have different spawn logic to maintain proper spacing

The game should now feel much more polished with logs that move in organized, evenly-spaced patterns without any overlapping or collision issues between logs in the same lane!

This is great, and exactly what I asked for. One thing to note is that Cursor does in a diff inside the IDE, much like other tools we have used. You can highlight each change or click Accept File to accept all changes. Let's see if we can add some functionality to this game (figure 10.10).

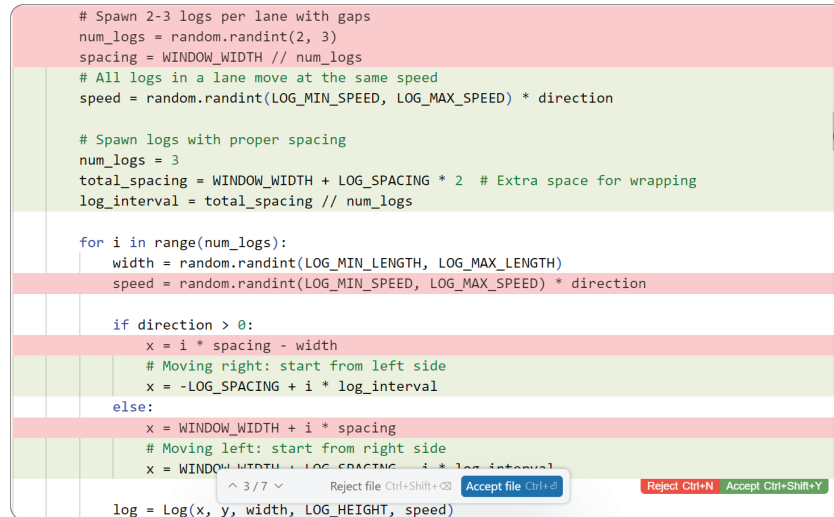


Figure 10.10 There is a diff interface that comes up in Cursor. You can accept individual changes or accept the entire file if you like.

Now that I've accepted the changes, it's time to run the application again. As figure 10.11 illustrates, the logs are now evenly spaced.

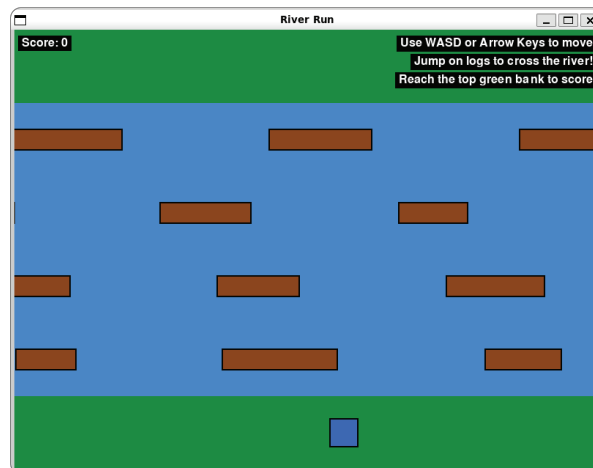


Figure 10.11 Now we can see the logs are evenly spaced and not overlapping.


This is great and exactly what I was looking for. Let's add some high scores. If you've ever played retro games, you know how important the high-score table used to be. Whether it was an arcade game that cost 25 cents, or an old console game, adding in your initials or something inappropriate was a lot of fun. Let's add that into our game. I know what I want from my high-score screen. Here are the objectives:

- I want a persistent scoreboard that lives on my hard drive, so the scores remain after I turn off the game.
- I don't want a database to store them.
- I want it to contain three characters and the score.
- I want it to be shown when the game starts up.

Let's use a high-quality prompt that will steer Cursor into generating exactly what we're looking for here. First, I want to talk about the structure I'm using for the prompt. I've found the following to be very effective in Cursor, Windsurf, and similar tools:

- *Goal*—What we want to achieve from the prompt
- *Gameplay*—Semantics about the gameplay to help understanding
- *Requirements*—Specific instructions for how we want things done
- *Flow*—A plain English explanation of the game flow

This helps the LLM in question understand what we're looking for and helps steer it toward the type of output we're looking for. This is what we've been doing extensively throughout the book, and it applies to vibe coding as well. Getting specific limits the creativity of the model when you don't want it to be creative. Here's the prompt that I created based on this structure:

 **Goal:** Add a persistent high-score system to this game without introducing new files except the JSON store.

Gameplay:

Each time the player crosses the river, they earn 1 point.

Higher numbers are always better.

When the highest score is reached, provide an input screen so the player can input their initials.

Initials will be three characters.

Requirements:

Storage – Persist scores in `high_scores.json` (same folder).

Keep the top 5 scores globally, allowing duplicate initials.

Create the file on first write. On IO error, print a warning and continue gameplay.

Main-menu integration – When the player is on the title screen, draw the high-score table beneath the “Press SPACE to start” prompt.

Flow:

When the player reaches the top bank

Load existing scores.

If `current_score` qualifies for the top 5:

Pause the game.

Display overlay “NEW HIGH SCORE! Enter initials:” plus a blinking cursor.

Capture exactly three alphanumeric keystrokes, auto-uppercase, allow BACKSPACE, commit on ENTER.

Update + save JSON.

Whether or not a record was set, show the refreshed table under the “Congratulations” banner for 3 seconds, then return to the main menu.

As figure 10.12 shows, we’re adding in the main file (`river_run.py`) as context. Cursor would likely assume this; however, it’s always good to include it. I’m also using the Agent mode and Claude 4 Sonnet. With this setup, we should be able to instruct Cursor to build exactly what we’re asking for.

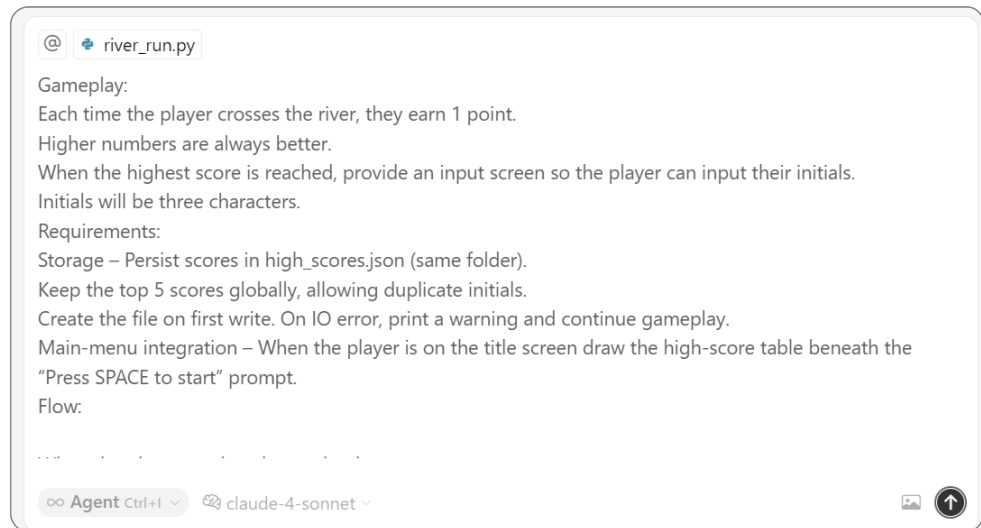


Figure 10.12 The exact configuration as we’ve sent using the main file as context, setting the Agent mode, and selecting Claude 4 Sonnet as our model

This prompt produced some interesting results. Rather than just coming back with a diff view of the changed code, the answer is more procedural. Cursor is showing me how it’s tackling this problem, step by step. As figure 10.13 illustrates, it makes a remark about the file being quite large. This is because, right now, the entire application is in a single file. Then, it reads the file and assumes the high-score system has already been implemented.

Here’s something new: it’s asking me to open the terminal and run the Python command to run the file. I’m going to skip it for now. The reason for that is, I’ve found that

using Cursor in Windows with WSL in the terminal is often problematic. Either Cursor will run Windows commands that don't work in WSL, or it won't properly read the output of the file and hang. It will say, "Tool call ended before result was received." This is true as of this writing and may be fixed in future versions. If you're running Cursor in Linux or MacOS, you won't have this problem.

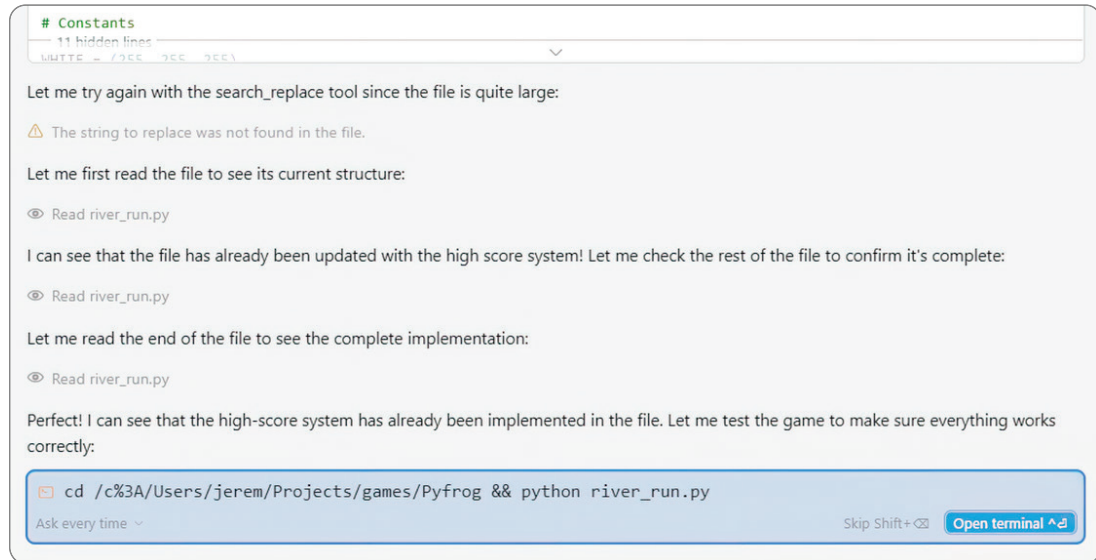


Figure 10.13 Cursor is reasoning its way to an answer and showing the steps taken. It looks like the large file is starting to cause some problems.

We'll skip the terminal (Shift + Enter), then run the command in the terminal ourselves, and see whether the changes were implemented for us. The first thing I noticed, in figure 10.14, we have a new startup screen. It asks us to press space to start, and it shows no high scores yet. This is because we have not yet played the game and built a high-scores table.

File sizes affect performance

We can see in our project, and as shown in figure 10.13, the search-and-replace tool can malfunction or slow down with large files. Suggestions also suffer sometimes when the files are too large. This likely has something to do with memory management, but there's an easy fix. You can refactor the code and break it into smaller files. This is good practice when developing software. If you're unsure how to break up the files, you can ask Cursor to do it for you—for example, use "Please refactor this code and break it into individual files for clarity and organization."



Figure 10.14
The gameplay of the game is changed based on our specification. We now see a menu screen that will display high scores once achieved.

I open the game and start playing. The gameplay itself has not changed at all. And as soon as I cross the river for the first time and get a point, I've achieved a high score. As figure 10.15 shows, I can now enter my initials and get added to the scoreboard.

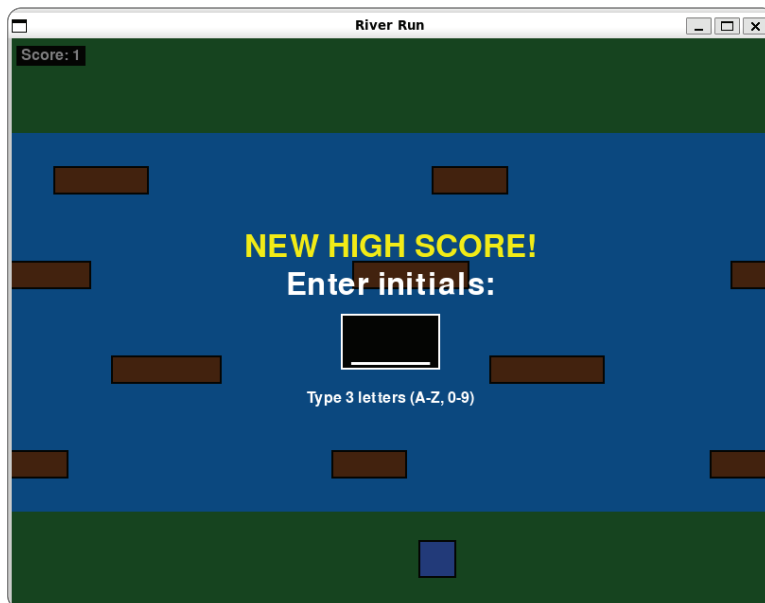


Figure 10.15
Cursor created the exact functionality we asked for. We now have a high score and input box to enter our initials.

Now I have entered my initials in and continue playing the game. I can now see the JSON file that was created by the application, where the scoreboard is held. Figure 10.16 shows the JSON structure. This JSON structure ensures we don't need to use a database, and we can do interesting things with JSON if we decide to in the future.



```
river_run.py  high_scores.json x
{} high_scores.json > ...
1  {
2    "scores": [
3      {
4        "initials": "JCM",
5        "score": 1
6      }
7    ]
8  }
```

Figure 10.16 The JSON structure the game has generated. It is a good lightweight storage method and a data structure we can use later if we want.

One thing I noticed is, after entering a score, it's immediately saved, and the interface returns to the main screen (figure 10.17). The only difference is I can see my high score now. This is exactly what I asked for, but I wasn't clear enough on the gameplay details.

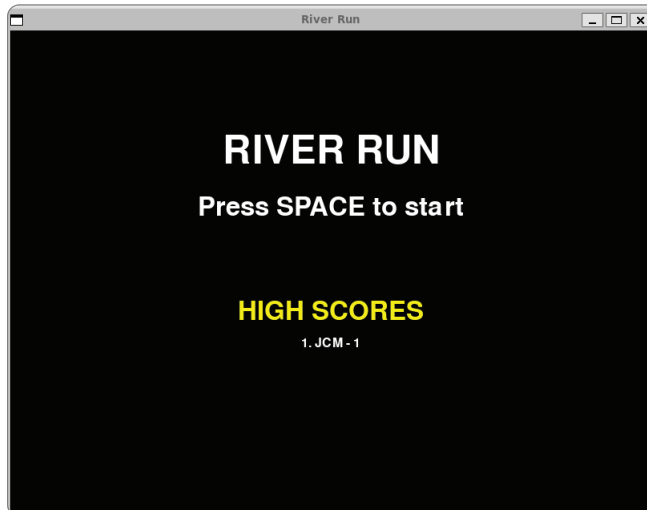


Figure 10.17 We are returned to the original menu screen. However, now we see our high-scores table with populated data

The problem we now have is that the game ends once the player reaches a high score. This means, of course, that every single person will get a high score of 1 every time they play. There is no way of getting a higher score than that. But there is an easy fix.

Here's the prompt I'll use:



This works as expected. However, when the player enters the high score, I would like to continue the game, allowing the player to try to cross river again, incrementing their score each time they successfully cross the river.

This is another “plain English” type prompt that explains what I’m trying to accomplish without having to get too technical. In the spirit of vibe coding in this case, we’re just asking Cursor to fix the problem, without worrying too much about implementation. Here is the output summary:



I'll modify the game flow so that after entering a high score, the player continues playing instead of returning to the title screen. This will allow for continuous gameplay where players can keep accumulating points. Now I need to update the game state management since we're no longer using the `SHOW_SCORES` state in the same way. Let me also add a way for players to return to the title screen when they want to quit their current session: Now let me update the UI to show the new quit instruction: Perfect! Now let me test the updated game to make sure the continuous gameplay works correctly:

```
cd /c%3A/Users/jerem/Projects/games/Pyfrog && python river_run.py
```

Great! I've successfully updated your River Run game to implement continuous gameplay. Here's what I changed:

Key Changes Made:

- 1 *Continuous Gameplay:* After entering initials for a high score, the game now continues playing instead of showing the score table and returning to the title screen.
- 2 *Seamless Score Accumulation:* When players cross the river and it's not a high score, the game continues without any interruption - no score display overlay.
- 3 *Added Quit Option:* Players can now press Q during gameplay to return to the title screen when they want to end their current session.
- 4 *Updated Instructions:* Added the new quit instruction to the on-screen help.

New Game Flow:

Player crosses river → Score increments → Player resets to starting position

If it's a high score → Initials input screen → Continue playing after entering initials

If it's not a high score → Just continue playing seamlessly

Press Q anytime → Return to title screen to see high scores or start fresh

This creates a much more engaging experience where players can build up impressive scores in a single session, with high score entry happening only when they beat their previous bests, but gameplay continuing uninterrupted afterward.

The high score system still works exactly as specified - it saves to `high_scores.json`, keeps the top 5 scores, handles IO errors gracefully, and shows the high score table on the title screen.

Judging by the output, it seems like Cursor understands what I meant. It not only makes changes to the code but explains the process it went through. Now I run the game, and it works exactly as expected. As shown in figure 10.18, the high-score screen comes up and allows me to enter my initials, then continue the game. This is exactly what I asked for.

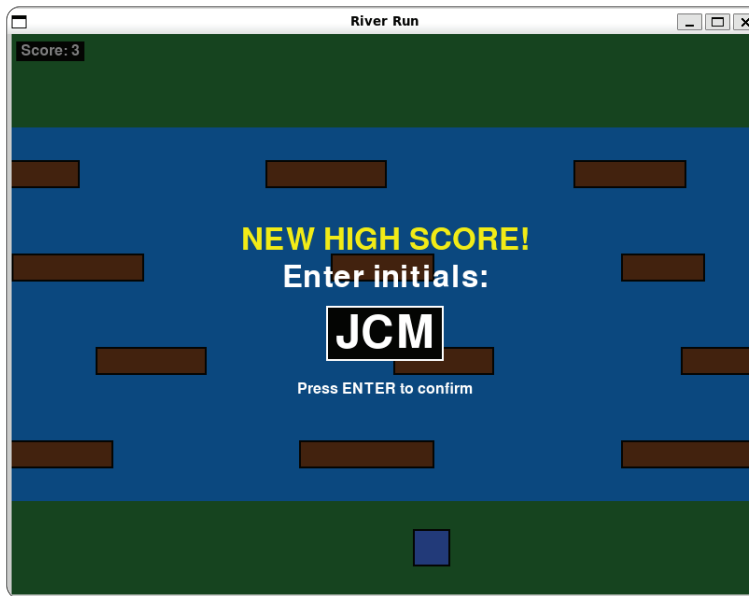


Figure 10.18 The game now allows you to enter your initials and continue game play and accumulate higher scores, which was not possible before.

When I start the game again, the scores persist from the JSON file, and I am shown top scores in the game. It isn't perfect yet, but we've seen how we can vibe code a working game quickly and easily. If you are specific and implement good prompt engineering, you can build a lot of impressive things with the vibe-coding approach.

Vibe coding with tools such as Cursor shows how AI can be a creative partner and a technical executor in software development. These tools mix natural language communication with strong agent capabilities, allowing for quick changes from ideas to working apps, as seen in our complete game made through chat prompts.

While these tools are favorites of the vibe-coding community, they're not limited to vibe-coding alone. As a professional developer, you can effectively use them in the same ways we've used other tools to this point. You can also vibe code with GitHub Copilot, Tabnine, and Blackbox AI as well.

Your success will hinge on when to embrace vibe coding's fluid, experimental nature and when to use more structured development practices. AI-powered development

tools are evolving. As we already know, they don't replace traditional programming skills but enhance them. This enables you to focus on creativity, problem-solving, and user experience while AI manages the mechanical parts of code generation.

The future of development isn't about choosing between human skills and AI help—it's about mastering the synergy between them.

Summary

- Vibe coding allows fast prototyping and experimentation through natural language with AI agents. It's great for proof-of-concepts, learning new tech, and creative projects where speed is key.
- Cursor's Agent mode offers autonomous development features. It can explore full codebases, make multifile changes, run tests, and handle complex workflow, all while keeping you informed on the process.
- Managing context effectively is key to success with AI tools. Including only relevant files and documentation helps avoid information overload. It also ensures the AI understands your project's needs and structure.
- High-quality initial prompts cut down iteration cycles. Clear specifications, technical limits, and desired outcomes help AI agents produce accurate and usable code right away.
- Professional developers can use vibe coding wisely in certain cases. They can stick with traditional structured methods for production systems, security-sensitive apps, and enterprise software, where accountability and thorough testing matter.

