Contents

Setting the Stage – VS Code vs. Cursor

Migrating from VS Code to Cursor

Choose Cursor if...	Choose VS Code + Copilot if...
You want the smartest AI right now. Cursor's integration with **Claude 3.5 Sonnet** and its specific UX features (Composer) generally outperform Copilot in complex reasoning.	**You work in a strict enterprise.** Many companies block Cursor for security reasons but have existing contracts for GitHub Copilot.
You want the AI to write entire features. Cursor's "Composer" mode is currently better at editing multiple files simultaneously to build a full feature at once.	**You want the lowest cost.** Copilot is ~$10/mo for individuals. Cursor Pro is $20/mo.
You are willing to switch editors. Even though it looks like VS Code, it is a separate installation.	**You want stability.** You prefer using the "official" editor with the massive Microsoft ecosystem backing it.

Since Cursor is built on the same foundation as VS Code, you don't have to lose your custom environment. You can bring your tools with you in one click.

From the Cursor App:
1. Open Settings: Press Ctrl + Shift + J (or Cmd + Shift + J on Mac) and navigate to the General tab.
2. Import Data: Locate the "Import from VS Code" section. Click Import.
3. Check Your Extensions: Press Ctrl + Shift + X. You should see your familiar extensions (like Prettier, Python, or Power Platform Tools) active and ready.
4. Sync Themes: Press Ctrl + K then Ctrl + T. Your preferred color scheme should now be available in the dropdown list.

AI Context Window - How much code can the AI actually read?

Think of the Context Window as the AI's "active attention span." If you give it too much irrelevant information, it loses focus; if you give it too little, it hallucinates.

How to Manage Your Context:
- **The Index:** When you first open a folder, watch the bottom right corner. Cursor "indexes" your files so it can find code across your whole project.
- **Manual Control with @:** Do not let the AI guess. Use the @ symbol to tell the AI exactly what to look at:
  - @Files: Pick specific files relevant to your current task.
  - @Folders: Give the AI the structure of a specific module.
  - @Codebase: Ask a question about the entire project (best for "Where is the login logic?").

Demonstration: Focused vs. Unfocused Prompts
- **Bad Prompt:** "Fix the error in my code." (The AI has to search everything).
- **Good Prompt:** "Look at @api.js and @schema.sql. Why is the POST request returning a 500 error?"

Privacy Mode vs. Business Mode

You must understand where your code goes. Cursor offers different levels of "forgetfulness" to ensure your intellectual property remains yours.

Configuring Your Privacy:

1. **Enter Privacy Mode:** In Settings (Ctrl + Shift + J), toggle **Privacy Mode** to **On**. This ensures your code is never used to train future AI models.
2. **Local Indexing:** Even with Privacy Mode on, Cursor creates a local "map" of your code on your hard drive to help you navigate. This data stays on your machine.
3. **Using .cursorignore:** If you have sensitive files (like .env files or banking API keys), create a file named .cursorignore in your project root. Add the filenames there to "blindfold" the AI from those specific files.

Practice Lab: Initial Setup

Perform these steps now to prepare your environment:

1. Import your VS Code settings.
2. Toggle Privacy Mode to "On."
3. Create a new folder named Banking_Demo.
4. Add a .cursorignore file and type config.json inside it.
5. Verify: Try to mention config.json in the Chat (Ctrl + L) using the @ symbol. If it doesn't appear, your privacy wall is working.

Introduction to Prompt Engineering for Code

The RTF Framework

To get professional results, you must treat the AI like a specialized colleague. Using the **RTF (Role, Task, Format)** framework prevents generic or "lazy" code.

- **Role:** Define the expertise level (e.g., "Senior Python Developer," "Security Auditor").
- **Task:** Define the specific action (e.g., "Refactor this loop," "Write unit tests").
- **Format:** Define how you want the answer (e.g., "A Git Diff," "A Markdown table," "Commented code").

Example Comparison:

- **Weak Prompt:** "Clean up this code."
- **RTF Prompt:** "Act as a **Senior React Developer**. **Refactor** the following component to use the useReducer hook instead of multiple useState calls. Output the result as a **Side-by-Side Diff**."

Scenario 1: A messy, "junior-level" JavaScript function that needs a professional refactor.

calculator.js:

```
function calc(a, b, type) {
  if (type == "add") {
    return a + b;
  } else if (type == "sub") {
      return a - b;
   } else {
      return "error";
   }
}
```

The Prompt to Demo:
"Act as a **Senior TypeScript Developer**. **Refactor** this function to use an enum for the operation types and ensure the function is strictly typed. **Output** the result as a modern TypeScript file."

Scenario 2: The "Logic & Security" Lab
Practice **RTF Prompting** to fix security flaws and logic gaps in a banking context.

Open this code and use Ctrl + K. Act as a **Senior Security Auditor** and ask the AI to fix the vulnerabilities.

```
// File: bankTransfer.js
function transferFunds(amount, accountId) {
   // BUG: No validation if amount is negative
   // BUG: No check if accountId exists
   console.log("Transferring " + amount + " to " + accountId);

   let balance = 1000;
   balance = balance - amount;

   return "Success! New balance: " + balance;
}
```

Recommended Prompt:
"Act as a **Senior Security Auditor**. **Refactor** this transfer function to: 1. Prevent negative transfers, 2. Ensure the accountId is a 10-digit string, and 3. Check for sufficient balance. **Output** the result as clean, production-ready code."

Chain-of-Thought Prompting

AI models often fail when they try to write 100 lines of code at once. **Chain-of-Thought** prompting forces the AI to "think out loud" and create a logical plan before it touches the keyboard. This significantly reduces "hallucinated" libraries or broken logic.

Step-by-Step Demo: The "Plan-First" Approach

1. Open **Chat (Ctrl + L)**.
2. Input a complex request but end it with a planning instruction:

*"I need a Node.js script that connects to a SQL database and exports transactions to a CSV. **Before writing any code, list the steps and logic you will follow. Wait for my approval.**"*

3. Review the AI's plan. If a step is wrong (e.g., it suggests the wrong library), correct it.
4. Type **"Proceed"** to have it generate the code based on the approved plan.

Scenario 1: Building a complex logic flow for a banking application (matching your interest in the financial sector).

The Prompt to Demo (In Ctrl + L Chat):
*"I need to build a 'Surprise Assumption of Duties' tracker for a bank. It needs to check if a manager has been on mandatory leave for 5 days, and if so, trigger a notification for a 'Surprise Assumption' by a relief officer. **Before writing any code, explain the logic flow and the database schema I would need. Do not write the code yet.**"*

Scenario 2: The "API & Async" Lab
Practice **Chain-of-Thought (CoT)** to handle asynchronous data fetching properly.

This code crashes because it doesn't wait for the data. Use Ctrl + L to ask the AI to **plan** the fix before writing it.

```
// File: userData.js
function getUserProfile(userId) {
  const data = fetch(`https://api.bank.com/users/${userId}`);
  // BUG: 'fetch' is async, but this code treats it as sync
  console.log(data.name);
  return data;
}
```

Recommended Prompt:
*"This code is failing because of asynchronous timing issues. **Before fixing it, explain the steps** needed to implement a proper async/await pattern with a try/catch block for error handling. Once I approve the plan, write the code."*

Zero-shot vs. Few-shot Prompting

Sometimes you want the AI to follow your specific coding "vibe"—like how you name variables or how you handle errors.

- **Zero-shot:** You ask for code with no examples. The AI uses its default style.
- **Few-shot:** You provide 1 or 2 examples of your preferred style first, then ask for the task.

Few-shot Demonstration:

1. Open **Inline Edit (Ctrl + K)**.
2. Provide context by pointing to your existing style:

"Here is how I handle errors in this project: try { ... } catch (e) { logError(e); }. Now, using that same style, write a function that fetches user data from @api.js."

3. Observe how the AI mirrors your exact error-handling pattern instead of using a generic console.log.

Scenario 1: Teaching the AI to follow a specific, custom error-handling pattern used in your classroom or company.

Copy this "Pattern Style" into a file named style_guide.js:

```
// OUR PROTECTED PATTERN:
// We always wrap in try-catch and use the custom 'BankLogger' class.
try {
  // logic here
} catch (err) {
  BankLogger.error("Process Failed", { timestamp: Date.now(), details: err });
}
```

The Prompt to Demo (In Ctrl + K while highlighting the code):
*"Using the **same pattern** shown here, write a new function called processLoanApplication that takes a user object and sends it to an API endpoint /v1/apply."*

Scenario 2: The "Style & Consistency" Lab
Practice **Few-shot Prompting** to match a specific UI component style.

Look at the "Pattern" component below. Use Ctrl + K to generate a NEW component (a NotificationCard) that looks exactly like it.

```
<div class="card shadow-sm border-blue">
  <h3 class="text-bold">Transaction Alert</h3>
  <p class="text-muted">Please review your recent activity.</p>
  <button class="btn-primary">View Details</button>
</div>
```

Recommended Prompt:
*"Using the **same HTML structure and CSS classes** (shadow-sm, text-bold, btn-primary) as the Transaction Alert card above, create a new NotificationCard. It should have a title for 'System Update' and a button that says 'Update Now'."*

The Interface – Chat (Cmd+L) vs. Inline Edit (Cmd+K)

Cmd+K (Inline)

The "Surgical" Tool

Use **Cmd+K** (or Ctrl+K on Windows) when you already know *where* the change needs to happen. It is designed for rapid-fire modifications within your active file. It doesn't just write code; it rewrites it and shows you a "diff" (comparison) so you can approve the changes.

Demonstration Examples:

1. **The "Modernizer":** Highlight an old JavaScript function and type: *"Convert this to an ES6 arrow function and use destructuring for the arguments."*
2. **The "Safety Net":** Highlight a block of database logic and type: *"Wrap this in a try-catch block and log the error to the console with a timestamp."*
3. **The "Language Swap":** Highlight a CSS block and type: *"Convert this standard CSS to Tailwind utility classes."*

Example: The "Refactor and Clean"
You have a "messy" function with deep nesting.
File: auth_service.js

```
function checkAccess(user) {
  if (user != null) {
    if (user.isLoggedIn) {
      if (user.role == 'admin') {
        return true;
      } else {
        return false;
      }
    } else {
      return false;
    }
  } else {
    return false;
  }
}
```

The Prompt (Ctrl+K): *"Use guard clauses to flatten this function and make it more readable."*

Example: Data Transformation
You have a list of raw data and need it formatted for a UI component.
File: data.js

```
const rawUsers = [
  { name: "John", age: 30, active: true },
  { name: "Jane", age: 25, active: false },
  { name: "Doe", age: 40, active: true }
];
```

The Prompt (Ctrl+K): *"Generate a function called 'formatUserLabels' that maps through this array and returns a string like 'NAME (ACTIVE)' for each entry."*

Example: Unit Test Generation
Adding tests to an existing utility. **File:** math_utils.js

```
export const calculateInterest = (principal, rate, years) => {
  return principal * Math.pow((1 + rate), years);
};
```

The Prompt (Ctrl+K): *"Generate 3 Jest unit tests for this function, covering standard input, zero years, and high interest rates."*

Cmd+L (Chat)

The "Consultant" Tool

Use **Cmd+L** (or Ctrl+L on Windows) when you are thinking about the "Why" or "Where" instead of the "How." This is your persistent sidekick that lives in a sidebar. It has access to your entire codebase and can explain complex relationships that span multiple files.

Demonstration Examples:

1. **The "Code Explorer":** Open a file you didn't write and ask: *"Explain how the data flows from this input field to the final database save. Which files are involved?"*
2. **The "Bug Hunter":** Paste an error message from your terminal into the chat and ask: *"I'm getting this 'NullReference' error. Looking at @Codebase, where am I most likely failing to initialize my variables?"*
3. **The "Brainstormer":** Ask: *"I need to add a 'Download PDF' feature to this bank report page. What libraries are already installed in this project that can help me do that?"*

Example: Structural Explanation
A student is lost in a new project.
Prompt (Ctrl+L): *"@Codebase Explain how the authentication middleware is applied to the routes. Which file defines the actual login logic?"*

Example: Debugging with Terminal Output
The code is throwing an error in the console.
Terminal Error: TypeError: Cannot read property 'map' of undefined at dashboard.js:15
Prompt (Ctrl+L): *"I am getting @Terminal Error in dashboard.js. Looking at the code in that file, why might the data variable be undefined when the component mounts?"*

Example: Architectural Planning
Deciding how to add a new feature.
Prompt (Ctrl+L): *"I want to add a 'Dark Mode' toggle to this app. Based on the current @Files (select your CSS and Main Component), what is the most efficient way to implement this using CSS variables?"*

Codebase Indexing

For the AI to be smart, it needs to "read" your whole project. This process is called **Indexing**. Without a proper index, the AI is just guessing based on the file you have open.

How to Manage Your Index:

1. **Check Status:** Look at the bottom of the Chat sidebar. You will see a small "Index" status.
2. **Force Indexing:** If you've added many new files and the AI seems "lost," go to **Cursor Settings > General > Codebase Indexing** and click **"Resync Index."**
3. **Computing Embeddings:** Cursor converts your code into "vectors" (mathematical representations). This allows you to ask questions like *"Where do we handle bank interest calculations?"* and the AI finds the logic even if the word "interest" isn't in the filename.

Example: Natural Language Search

Instead of using Ctrl+F to find a specific word, you can ask Chat:

Prompt (Ctrl+L): *"Where in this project do we handle the logic for calculating the 'Surprise Assumption' dates for the banking staff?"*

Even if the student doesn't know the filename, the AI will navigate to the correct logic because it understands the *meaning* of the code.

Example: Finding Redundancy

Prompt (Ctrl+L): *"Scan @Codebase and tell me if there are any duplicate utility functions for date formatting. If so, which one should I keep?"*

Context Management

@Files & @Folders

Instead of copy-pasting code into a chat, you use @Files to point the AI to specific logic. This is essential for fixing bugs where the error is in one file, but the cause is in another.

Steps:

1. **Open Chat:** Press Ctrl + L.
2. **Trigger Mention:** Type the @ symbol. A dropdown will appear.
3. **Select Target:** Type the name of a file or folder and select it.
4. **Prompt:** Ask your question based on those specific files.

Use-Case Examples:

- **Dependency Mapping:** *"Explain how @login.js uses the functions defined in @auth-utils.js."*
- **Style Porting:** *"Look at @header.css. Create a new @footer.css that uses the same color scheme and font variables."*
- **Data Flow:** *"I'm sending a request from @form.js. Show me how it is handled in the @api/submit folder."*
- **Folder Summary:** *"Give me a high-level summary of all the logic contained in the @services folder."*
- **Logic Sync:** *"Ensure the variable names in @validation.js match the database schema in @schema.sql."*

@Docs

AI models are trained on data that is months or years old. If you are using a brand-new library or a specific API (like a banking API), the AI might hallucinate old syntax. @Docs allows you to feed the AI the *latest* documentation in real-time.

Steps:

1. **Add Documentation:** Type @Docs > **"Add new doc"**.
2. **Paste URL:** Paste the URL of the official documentation (e.g., https://tailwindcss.com/docs).
3. **Wait for Indexing:** Cursor will crawl the site to learn the library.
4. **Prompt:** Use the doc name in your query.

Examples:

- **New Syntax:** *"Using @Tailwind CSS, create a grid layout that doesn't exist in older versions of CSS."*
- **API Integration:** *"Using @Stripe Docs, write a Node.js function to create a recurring subscription checkout session."*
- **Component Library:** *"Build a 'Data Table' using @Shadcn UI that includes pagination and sorting."*
- **Framework Updates:** *"Convert this @Next.js Page Router code to the new App Router structure using the @Next.js Docs."*
- **Library Specifics:** *"Using @Zod, create a validation schema for a user registration form that requires a strong password."*

@Codebase

@Codebase is the "heavy lifter." It tells Cursor to look through every single file you have indexed. Use this for architectural questions or when you have no idea where a specific piece of code lives.

Steps:

1. **Open Chat:** Press Ctrl + L.
2. **Select Codebase:** Type @Codebase or click the **"Codebase"** button above the input box.
3. **Ask High-Level Questions:** Focus on the "big picture."

Examples:

- **The "New Hire" Query:** *"@Codebase Explain the overall architecture of this project. Where does the data start and where does it end up?"*
- **Impact Analysis:** *"If I change the 'Interest Rate' variable in the central config, which files across the @Codebase will be affected?"*
- **Bug Investigation:** *"I am getting a 'Connection Timeout' error intermittently. Search the @Codebase for every place we connect to an external database."*
- **Feature Inventory:** *"Does this project already have a utility for converting currency? Search @Codebase before I write a new one."*
- **Security Audit:** *"Search the @Codebase for any hardcoded API keys or sensitive credentials that should be moved to an .env file."*

Cursor Composer (Cmd+I)

Standard Chat vs "Composer" mode

While Chat (Cmd+L) is great for asking questions and copying code snippets, it cannot *apply* changes to multiple files at once. **Composer (Cmd+I)** opens a dedicated floating window (or full screen) where the AI generates the code and immediately stages it across your entire workspace for your approval.

Steps using Composer

1. **Trigger Composer:** Press Cmd+I (Mac) or Ctrl+I (Windows).
2. **Choose Mode:** Select **"Normal"** (fast, one-shot generation) or **"Agent"** (the AI runs terminal commands and iteratively fixes its own errors).
3. **Provide Context:** Use the @ symbol to attach necessary files.
4. **Prompt:** Tell the AI exactly what to build across the system.
5. **Accept/Reject:** Review the multi-file diffs and click "Accept All" (or Cmd+Enter).

Examples:

1. **The "Global Theme Shift":** *"Change our primary brand color from blue to emerald green. Update the tailwind.config.js, the @Button.jsx component, and the @header.css file simultaneously."*
2. **The "API Version Bump":** *"We are moving from API v1 to v2. Find every fetch request in the @services folder and update the base URL, then update the expected JSON response payload in our frontend interfaces."*
3. **The "Mass Refactor":** *"Rename the Customer data model to Client across the entire project. Update the database schema, the backend controllers, and the frontend state management."*
4. **The "Boilerplate Generator":** *"Set up a new React project structure. Create folders for components, pages, and hooks. Give me a basic index.js and App.js with a dark mode toggle to start."*
5. **The "Cross-File Debugging":** *"When a user logs out, their token isn't clearing properly. Fix the logout function in @auth.js, ensure the token is removed from localStorage in @navbar.jsx, and clear the backend session in @routes.py."*

Scaffolding a new feature

You can use Composer to generate the frontend UI, the backend API route, and the database schema in a single, well-crafted prompt. This requires combining the **RTF Framework** and **Chain-of-Thought.**

Steps

1. **Open Composer (Ctrl+I).**
2. **Set the Rules:** Define the stack (e.g., React, Node, SQL).
3. **Define the Layers:** Explicitly ask for the Database, Backend, and Frontend.
4. **Review File Creation:** Watch Composer create entirely *new* files in your sidebar automatically.

Examples

1. **The Authentication Flow:** *"Build a user login feature. 1. Create a users.sql table schema. 2. Create an authController.js with bcrypt hashing. 3. Create a LoginForm.jsx with email/password validation. Wire them together."*

2. **The "Surprise Assumption" Audit Tool:** *"Build a dashboard for HR to track mandatory banking leave. Create the SQL schema for LeaveSchedules, an API route to fetch employees on leave for >5 days, and a UI table component to display them with a red 'Flag' button."*
3. **The CSV Exporter:** *"I need a feature to export transaction histories. Create a backend route that converts a JSON array to CSV, and a frontend ExportButton.jsx component that triggers the download when clicked."*
4. **The Support Ticketing System:** *"Scaffold a helpdesk ticket system. Create a Ticket Prisma schema (id, issue, status). Write the Next.js API route to POST a new ticket, and build the Tailwind-styled form for the user to submit it."*
5. **The Live Search Bar:** *"Create an auto-completing search bar for our employee directory. Build the frontend input component with a 300ms debounce, and the backend SQL query that uses the LIKE operator to search names."*

Reviewing "Diffs"

The AI is incredibly fast, but it is not infallible. When Composer modifies 8 files at once, you must audit the changes before hitting "Accept All." A "Diff" (Difference) shows deleted code in **Red** and new code in **Green**.

Steps

1. **Do Not Blindly Accept:** When Composer finishes, a list of modified files appears. Click on each file one by one.
2. **Scan for Red Flags:** Look at the red lines. Did the AI delete a custom workaround or a specific comment you needed?
3. **Partial Acceptance:** If 90% of a file is good, but one block is wrong, you can highlight the bad code and ask Composer to fix just that part before accepting the whole file.
4. **Reject File by File:** You can click the "X" on a specific file to discard the AI's changes for that file while keeping the changes in the others.

Examples:

1. **The "Destructive Overwrite":** The AI rewrites a complex, highly optimized SQL query with a generic, slower one because it didn't fully understand your business logic. *Action: Reject the diff for that file.*
2. **The "Hallucinated Import":** In the green text, you see import { MagicTool } from 'magic-library'. You do not have this library installed. *Action: Tell Composer, "I don't have magic-library installed, write this using vanilla JavaScript instead."*
3. **The "Orphaned State":** The AI successfully adds a new column to your database schema and updates the backend, but in the frontend diff, you notice it forgot to add the new field to the UI form. *Action: Prompt Composer to add the missing field to the frontend component.*
4. **The "Security Leak":** The AI hardcodes a database password directly into the green code instead of using process.env.DB_PASSWORD. *Action: Highlight the line and ask it to use environment variables instead.*
5. **The "Collateral Damage":** The AI fixes your navigation bar, but accidentally deletes the CSS class that was keeping your logo centered. *Action: Use the inline diff editor to restore the missing CSS class before accepting.*

Debugging & Self-Correction Loops

"Fix with Cursor" in Terminal

When a script crashes, the terminal spits out a wall of red text. Instead of copying that text, opening a browser, and searching StackOverflow, Cursor can read the terminal directly, understand the context of the crash, and write the fix in the exact file that caused it.

Steps

1. **Run the Code:** Execute your script (e.g., npm start or python main.py) in the Cursor integrated terminal.
2. **Locate the Error:** When the code crashes, look for the blue **"Fix with AI"** button that appears next to the stack trace in the terminal.
3. **Trigger the Fix:** Click the button (or highlight the error text and press Cmd+L).
4. **Review the Diff:** Cursor will open Chat, explain *why* the error happened, and generate an inline edit block to fix the offending file.
5. **Apply:** Click "Apply to file" and accept the changes.

Examples:

1. **The Missing Dependency:** * *Error:* Error: Cannot find module 'bcrypt'
   - *AI Action:* The AI explains the module is missing and provides the terminal command to install it (npm install bcrypt).

2. **The Port Conflict:**
   - *Error:* EADDRINUSE: address already in use :::8080
   - *AI Action:* The AI identifies the server conflict and modifies your server.js to automatically try port 8081 if 8080 is busy.

3. **The Type Mismatch:**
   - *Error:* TypeError: Assignment to constant variable.
   - *AI Action:* The AI locates the specific const declaration in your JavaScript file and changes it to let.

4. **The Database Timeout:**
   - *Error:* MongoTimeoutError: Server selection timed out after 30000 ms
   - *AI Action:* The AI checks your .env connection string, suggests checking your IP whitelist, and adds retry logic to your database connector file.

5. **The Syntax Typos:**
   - *Error:* SyntaxError: Unexpected token } in JSON at position 145
   - *AI Action:* The AI finds the malformed JSON file in your project and removes the trailing comma causing the issue.

Iterative Prompting

AI models are not perfect. Sometimes they get stuck in a "loop" (making the same mistake repeatedly) or they hallucinate libraries that don't exist. The "Refusal Strategy" teaches you to stop being polite and start being incredibly strict by telling the AI exactly what *not* to do.

Steps

1. **Identify the Hallucination:** Notice that the AI's generated code is failing for the same reason twice.
2. **Stop the Generation:** Press Esc to immediately stop the AI from typing.
3. **State the Failure Explicitly:** Tell the AI exactly what line failed.
4. **Apply a Hard Constraint (The Refusal):** Use the word "Do NOT" or "Refusal" to fence the AI in.
5. **Regenerate:** Press Enter to have the AI try a completely new approach.

Examples:

1. **Refusing Deprecated Code:** *"You keep using the old v5 syntax for React Router.* ***Do NOT*** *use useHistory. You must use useNavigate from v6."*
2. **Refusing Hallucinated APIs:** *"You hallucinated the getBankRoles() endpoint. That does not exist in our system. Look at @api-routes.js and only use the endpoints listed there."*
3. **Refusing Over-Complication:** *"This Regex you wrote for email validation is too complex and is failing on international domains.* ***Do NOT*** *use Regex. Write a standard string manipulation function instead."*
4. **Refusing Destructive Edits:** *"When you refactored that function, you deleted my JSDoc comments.* ***Do NOT*** *remove any existing comments when optimizing code. Rewrite it and put the comments back."*
5. **Refusing Third-Party Libraries:** *"I cannot install new NPM packages for this enterprise project.* ***Do NOT*** *suggest using lodash. Write the array sorting logic using only vanilla JavaScript."*

Linter Integration

Linters (like ESLint or Prettier) enforce coding standards, which is crucial in team environments. Instead of manually fixing 50 missing semicolons or variable naming warnings, you can pipe your linter output directly to the AI and have it fix the entire codebase in seconds.

Steps

1. **Run the Linter:** Run your linting script in the terminal (e.g., npm run lint).
2. **Capture the Output:** Highlight the list of warnings in the terminal and press Cmd+L.
3. **Formulate the Bulk Prompt:** Instruct the AI to resolve a specific rule violation across the affected files.
4. **Use Composer (Optional):** If the warnings span more than 3 files, press Cmd+I (Composer) to let the AI stage multi-file edits.
5. **Audit and Accept:** Review the diffs carefully to ensure the logic wasn't altered while fixing the syntax.

Examples:

1. **The 'Any' Type Purge:** *"The linter is throwing 12 warnings for @typescript-eslint/no-explicit-any. Look at @Codebase and replace all any types with proper interfaces."*
2. **The Dependency Array Fix:** *"React is warning about missing dependencies in useEffect. Review @dashboard.jsx and add the missing variables to the dependency array without causing an infinite loop."*
3. **The Global Renaming (CamelCase):** *"We have linter warnings for snake_case variables. Go through the @services folder and convert all snake_case variables to camelCase to satisfy the linter."*
4. **The Unused Import Cleanup:** *"My terminal shows 20 warnings for unused imports. Open Composer (Cmd+I) and safely remove all unused import statements across the entire @components folder."*
5. **The Callback Conversion:** *"The linter prefers arrow functions (prefer-arrow-callback). Find all standard function() declarations used as callbacks in @api.js and convert them to arrow functions."*

Refactoring & Legacy Code Modernization

Strategy: "Explain, Plan, Refactor"

When you encounter a wall of undocumented, 10-year-old code, the worst thing you can do is highlight it and tell the AI to "make it better." The AI might optimize away critical, hidden business logic. The "Explain, Plan, Refactor" strategy forces the AI to prove it understands the code before it touches it.

Steps:

1. **Explain:** Highlight the legacy block, press Cmd+L (Chat), and ask: *"Explain what this code does step-by-step. Identify any edge cases it handles."*
2. **Verify:** Read the explanation. If the AI missed something, correct it in the chat.
3. **Plan:** Prompt: *"Now, create a plan to refactor this to be more modular and readable. Do not write the code yet."*
4. **Refactor:** Once you approve the plan, press Cmd+I (Composer) or use Cmd+K (Inline) and prompt: *"Execute the approved refactoring plan."*

Examples:

1. **The Nested Nightmare:** * *Legacy:* A function with 6 levels of nested if/else statements.
   - *Refactor:* Ask the AI to explain the logic paths, then refactor using "early returns" (guard clauses) to flatten it.
2. **The "Magic Number" Script:** * *Legacy:* An old data-processing script full of hardcoded numbers (e.g., x * 0.12).
   - *Refactor:* Have the AI identify what the numbers mean, plan a configuration object, and extract the numbers into named constants (e.g., TAX_RATE).
3. **The Monolithic Sysadmin Script:** * *Legacy:* A massive 500-line Bash script used for server backups and cron jobs.

    - *Refactor:* Ask the AI to explain the sequence, plan a modular approach, and refactor it into a modern, error-handled Python automation script.
4. **The Undocumented Database Query:** * *Legacy:* A massive raw SQL string with multiple JOINs and no comments.
    - *Refactor:* Have the AI translate the SQL into plain English, then rewrite it using a modern ORM (like Prisma or Entity Framework).
5. **The Spaghetti jQuery:** * *Legacy:* An old HTML file tightly coupled with jQuery DOM manipulations.
    - *Refactor:* Ask the AI to map the state changes, then plan and execute a conversion into a clean React or Vue component.

Converting syntaxes

Migrating a codebase from one language or framework to another is tedious. Cursor excels at this, but it needs a "North Star" to follow. By using the @ symbol to reference an already-converted file, you ensure the AI matches your exact architectural style during the conversion.

Steps:

1. **Open Target:** Open the old file you want to convert.
2. **Provide Reference:** Press Cmd+K (Inline Edit). Type @ and select a file that represents your *new* standard (e.g., @Button.tsx as your TypeScript standard).
3. **Prompt the Conversion:** Instruct the AI to convert the current file to match the syntax and patterns of the reference file.
4. **Audit:** Review the diff to ensure library imports and syntax match the new standard.

Examples:

1. **JavaScript to TypeScript:** * *Prompt: "Convert this .js file to .ts. Look at @types.ts to see how we define interfaces, and apply strict typing to all function parameters."*
2. **Vanilla CSS to Tailwind CSS:** * *Prompt: "Replace all custom CSS classes in this HTML file with @Tailwind utility classes. Remove the associated <style> block."*
3. **Legacy OS Firewalls:** * *Prompt: "Convert these old CentOS 6 iptables routing rules into modern RHEL 9 firewalld commands. Output as a shell script."*
4. **React Classes to Hooks:** * *Prompt: "Convert this React Class Component into a Functional Component. Look at @Header.jsx to see how we use useEffect and useState."*
5. **REST API to GraphQL:** * *Prompt: "We are migrating our data fetching. Look at @UserQuery.graphql for our schema style, and convert this standard fetch() REST call into an Apollo GraphQL useQuery implementation."*

Creating "Cursor Rules"

If you find yourself constantly typing *"Always use arrow functions"* or *"Don't use console.log,"* you are wasting time. You can enforce team standards automatically by creating a .cursorrules file. This acts as a permanent, invisible system prompt for every AI interaction in that specific project folder.

Steps:

1. **Create the File:** In the root directory of your project, create a new file named exactly .cursorrules.
2. **Define the Persona:** Start by telling the AI how to act (e.g., "You are an expert enterprise developer...").

3. **List the Rules:** Write your strict coding standards in plain, clear English. Use bullet points.
4. **Test the Enforcement:** Open a new file, press Cmd+K, and ask it to write a simple function. Verify it followed the hidden rules.

Examples:

1. **Formatting Rule:** *"Always format monetary outputs in Philippine Peso (PHP) using the Intl.NumberFormat API, regardless of the placeholder data."*
2. **Error Handling Rule:** *"Never use console.log for errors. You must always import our custom logger from @utils/logger.js and use Logger.error()."*
3. **Syntax Rule:** *"Always use ES6 arrow functions. Never use the function keyword unless defining a React Component."*
4. **Testing Rule:** *"Whenever generating a new utility function, you must automatically create a sibling file with basic Vitest unit tests."*
5. **Security Rule:** *"Never hardcode API URLs or secrets. Always assume they come from process.env. If an environment variable is missing in your logic, add a comment to remind the developer."*

Documentation & Test Generation

Auto-generating docstrings and README.md

Writing good documentation is critical for team collaboration, but it is often neglected. Cursor can read existing code and generate perfectly formatted docstrings, comments, and even comprehensive README.md files in seconds.

Steps:

1. **For Inline Docs:** Highlight a complex function or class.
2. **Trigger Edit:** Press Cmd+K (Inline Edit).
3. **Prompt:** *"Add JSDoc (or Python Docstring) comments to this function explaining the parameters, return types, and purpose."*
4. **For READMEs:** Open Chat (Cmd+L) or Composer (Cmd+I).
5. **Global Prompt:** *"Review @Codebase and generate a professional README.md that includes setup instructions, required environment variables, and a brief overview of the architecture."*

Examples:

1. **The JSDoc Standard:** Highlight a JavaScript utility function. *Prompt: "Add standard JSDoc comments detailing the @param types and @returns value."*
2. **The Python Sphinx Style:** Highlight a Python class. *Prompt: "Generate Google-style docstrings for this class and all its methods."*
3. **The "Explain Like I'm 5" Comment:** Highlight a confusing Regex pattern or bitwise operation. *Prompt: "Add a plain-English inline comment above this line explaining exactly what this pattern matches."*
4. **The API Swagger Spec:** Highlight an Express.js or FastAPI route. *Prompt: "Generate an OpenAPI/Swagger YAML comment block above this route describing the request body and potential 200/400/500 responses."*

5. **The Instant README:** Open Composer. *Prompt: "Create a README.md for this project. Extract the script commands from @package.json and create a 'How to Run' section."*

Test-Driven Development (AI-Style)

Traditional TDD involves writing a failing test, then writing the code to make it pass. With AI, you write the test (or have the AI write it), and then instruct Cursor to write the implementation until the test suite turns green. This guarantees the AI writes code that meets your exact business requirements.

Steps:

1. **Write the Test First:** Create a new file (e.g., calculator.test.js) and define the expected behavior.
2. **Create the Target File:** Create an empty calculator.js file.
3. **Open Composer (Cmd+I):** Switch to **Agent Mode**.
4. **The TDD Prompt:** *"Look at @calculator.test.js. Implement the functions in @calculator.js. Run the tests in the terminal (npm test) and iterate on your code until all tests pass."*
5. **Watch the Agent:** The AI will write the code, run the terminal, see the errors, and rewrite the code automatically.

Examples:

1. **The Pure Logic Test:** *Test:* A currency formatter must add commas and the PHP symbol. *Prompt: "Write the implementation in formatters.js to make this currency test pass."*
2. **The Edge-Case Test:** *Test:* A division function must throw a specific "DivideByZeroError" when dividing by 0. *Prompt: "Implement the division logic so it passes the standard math test and explicitly passes the zero-division error test."*
3. **The UI Component Render Test:** *Test:* A React button must display a loading spinner when the isLoading prop is true. *Prompt: "Build the Button.jsx component to satisfy this React Testing Library suite."*
4. **The API Status Code Test:** *Test:* A mock API test expects a 401 Unauthorized if the token is missing. *Prompt: "Write the Express middleware to check for the Bearer token and pass this supertest suite."*
5. **The Database Mock Test:** *Test:* A test checking if a user lookup returns null when an ID doesn't exist. *Prompt: "Implement the Prisma database query to satisfy this mock data test."*

Edge Case Discovery

Developers usually test the "happy path" (where everything goes right). The AI is exceptional at finding the "sad path" (where things break). You can use Cursor as a security auditor or QA tester by asking it to actively try and break your logic.

Steps:

1. **Highlight Target Code:** Select the function or module you want to stress-test.
2. **Open Chat (Cmd+L):** 3. **The "Break It" Prompt:** *"Act as a strict QA engineer. What inputs or scenarios would cause this function to fail, crash, or return incorrect data?"*
3. **Review the Findings:** The AI will list out edge cases (e.g., null values, negative numbers, extreme lengths).

4. **Apply the Fix:** Ask the AI: *"Refactor the code using Cmd+K to protect against items 1, 2, and 4 on your list."*

Examples:

1. **The Age Calculator:** *Code:* currentYear - birthYear. *Prompt: "What breaks this?" AI Finding:* Leap years, timezone differences, birth years in the future, non-integer inputs.
2. **The Array Reducer:** *Code:* data.reduce((a, b) => a.price + b.price). *Prompt: "What inputs break this?" AI Finding:* Passing an empty array (throws an error), passing objects without a price key (results in NaN).
3. **The User Input Form:** *Code:* Saving a username directly to the database. *Prompt: "Find security edge cases." AI Finding:* SQL Injection strings, XSS script tags, names exceeding database column limits, emojis breaking encoding.
4. **The Async Fetcher:** *Code:* await fetch(url). *Prompt: "What environmental edge cases will crash this?" AI Finding:* Network disconnects, DNS resolution failures, the server returning a 500 HTML page instead of JSON.
5. **The Financial Math:** *Code:* amount * 0.05. *Prompt: "What inputs would break this banking logic?" AI Finding:* Floating-point precision errors (e.g., 0.1 + 0.2 = 0.30000000000000004), negative deposit amounts.

Mini-Project Sprint – "The 1-Hour MVP"

Defining the tech stack and project structure via prompt

The blank canvas is often the most intimidating part of development. Instead of spending an hour running npm install, configuring Webpack, and creating folder structures manually, you will use Cursor's Composer to scaffold the entire foundation in one prompt.

Steps

1. **Open an Empty Folder:** Start with a completely blank directory in Cursor.
2. **Open Composer (Cmd+I):** Ensure you are in "Agent" mode so it can run terminal commands.
3. **Define the Stack:** Explicitly state the frontend, backend, and database technologies.
4. **Define the Structure:** Tell the AI what folders you need (e.g., components, pages, api).
5. **Execute:** Watch the AI create the files, populate the boilerplate code, and even run the installation commands in the terminal.

Examples:

1. **The Next.js Todo App:** *"Scaffold a Next.js Todo app with Supabase. Create a components folder for the UI, an actions folder for the database logic, and initialize Tailwind CSS for styling. Run the necessary install commands."*
2. **The Python Banking API:** *"Create a FastAPI backend with SQLite. Scaffold the project with a routers folder for endpoints, a models.py for SQLAlchemy schemas, and a main.py entry point. Include a requirements.txt."*
3. **The Vue Admin Dashboard:** *"Scaffold a Vue 3 project using Vite. Include Vue Router and Pinia for state management. Create a basic folder structure with views, components, and store."*

4. **The React Native Tracker:** *"Initialize an Expo React Native app for an Expense Tracker. Scaffold a screens folder, a components folder, and set up a basic navigation stack using React Navigation."*
5. **The Express CRM Backend:** *"Build a Node.js Express server configured with MongoDB (Mongoose). Create folders for controllers, routes, and models. Generate a basic .env.example file."*

Building the UI using "Image-to-Code"

Writing CSS and HTML layouts from scratch is time-consuming. Cursor includes a Vision model that allows you to upload a screenshot of a design, a wireframe, or even a hand-drawn sketch, and the AI will generate the exact code to replicate it visually.

Steps:

1. **Capture the Design:** Take a screenshot of a UI you want to build (from Dribbble, Figma, or a competitor's site).
2. **Attach the Image:** Open Chat (Cmd+L) or Composer (Cmd+I) and drag-and-drop the image into the input box (or paste it from your clipboard).
3. **Specify the Output:** Tell the AI exactly what framework and CSS system to use.
4. **Generate and Refine:** Let the AI write the code. If the margins or colors are slightly off, use Cmd+K to tweak them.

Examples:

1. **The Pricing Table Screenshot:** *Attach image. "Recreate this pricing table exactly as shown using React and Tailwind CSS. Ensure it is fully responsive for mobile screens."*
2. **The Hand-Drawn Login Form:** *Attach sketch. "Convert this hand-drawn wireframe into a clean, modern HTML/CSS login page. Use a dark mode color palette with a blue primary button."*
3. **The Settings Screen Mockup:** *Attach Figma export. "Build this settings dashboard using Vue.js. Extract the icons shown in the image and use placeholder SVG icons from Heroicons in their place."*
4. **The Data Card Dashboard:** *Attach design. "Recreate this financial metrics card using pure CSS. Make sure the 'trend up' indicator uses the exact shade of green shown in the image."*
5. **The Mobile App Navigation:** *Attach screenshot. "Build this bottom navigation bar for a React Native app. Ensure the active tab indicator accurately reflects the styling in the screenshot."*

Connecting the logic and state management using Composer.

A beautiful UI is useless if it doesn't do anything. The final iteration of the MVP involves taking those static, generated components and wiring them up to your backend, database, or global state. Composer is ideal here because it can look at your frontend UI and your backend routes simultaneously and bridge the gap.

Steps:

1. **Open Composer (Cmd+I).**
2. **Tag the Ends:** Use @Files to tag the static UI component (e.g., @LoginForm.jsx) and the logic handler (e.g., @auth-api.js).
3. **Define the Action:** Instruct the AI to replace static dummy data with live state and handle the user interactions (clicks, submits).

4. **Review the Diff:** Carefully check that the AI correctly mapped the input fields to your database schema.

Examples:

1. **Wiring the Todo UI to Supabase:** *"Look at @TodoUI.jsx and @supabaseClient.js. Connect the 'Add Task' button so it inserts a new row into the Supabase database, and update the UI state so the new task appears immediately without refreshing."*
2. **Connecting the Login Form:** *"Wire @LoginForm.html to @auth-controller.js. When the user submits the form, send a POST request. If it returns a 200, redirect to /dashboard. If it fails, display the error message in the red text span."*
3. **Implementing a Shopping Cart State:** *"Look at the static @ProductCard.vue and the @cartStore.js Pinia store. When 'Add to Cart' is clicked, dispatch the item to the store and update the badge count on the @Navbar.vue."*
4. **Hooking Up Live API Search:** *"Connect the static @SearchBar.tsx to the @employeeDatabase.sql schema. Implement a debounce function so that as the user types, it fetches live results from the backend and populates the dropdown."*
5. **Form Validation and Error Handling:** *"Look at the @CheckoutForm.jsx. Before it submits data to @stripe-api.js, implement client-side validation state. Highlight the input borders in red if the fields are empty or if the email format is invalid."*

AI as a Reviewer – Code Quality Assurance

Using Cursor to conduct a "Security Audit" on your code

AI models are trained on thousands of security vulnerabilities, including the OWASP Top 10. Instead of waiting for a dedicated security team to flag an issue, you can use Cursor to proactively scan your logic for SQL injections, command injections, or accidentally exposed secrets.

Steps:

1. **Open Target Files:** Open the files that handle user input, database queries, or external system commands.
2. **Trigger Chat (Cmd+L):** Add the relevant files using @Files (e.g., @authController.js and @db.js).
3. **Assign the Persona:** Prompt: *"Act as an expert Cybersecurity Auditor."*
4. **Define the Task:** *"Review the attached files for any security vulnerabilities, including injection flaws, exposed credentials, or improper access controls. List your findings and rank them by severity."*
5. **Execute Fixes:** Once vulnerabilities are found, use Cmd+K (Inline Edit) to apply the AI's suggested patches (like parameterized queries or environment variable integration).

Examples:

1. **The SQL Injection Trap:** *Code:* SELECT * FROM users WHERE username = ' + userInput + '. *Prompt: "Identify the injection vulnerability here and rewrite it using parameterized queries."*
2. **The Hardcoded Secret:** *Code:* An Azure connection string or SSH key pasted directly into a configuration file. *Prompt: "Audit this file for exposed credentials. Refactor it to securely load these values from a .env file."*

3. **The Command Injection (Sysadmin Script):** *Code:* A Node.js script that takes user input and executes a raw Linux shell command (e.g., exec('ping ' + targetIP)). *Prompt: "How could a malicious user exploit this shell execution? Refactor it to sanitize the input."*
4. **Insecure Direct Object Reference (IDOR):** *Code:* An endpoint fetching a financial record purely based on a URL parameter (e.g., /api/records/1234). *Prompt: "Check this route for IDOR vulnerabilities. Add middleware to verify the requesting user actually owns the record being accessed."*
5. **Cross-Site Scripting (XSS):** *Code:* A React component dangerously setting inner HTML from an unfiltered user comment. *Prompt: "Audit this component for XSS risks and implement a sanitization library to neutralize malicious scripts."*

Performance Optimization

Sometimes code works perfectly but runs terribly. Cursor is excellent at algorithmic analysis. It can identify mathematically inefficient code—like $O(n^2)$ nested loops—and suggest optimizations that run in $O(n \log n)$ or $O(1)$ time, which is critical when processing large datasets.

Steps:

1. **Locate the Bottleneck:** Identify the function or script that is running slowly.
2. **Highlight and Prompt (Cmd+K or Cmd+L):** Select the code block.
3. **Ask for Analysis:** *"Analyze the time and space complexity of this function using Big O notation. Why is it slow?"*
4. **Request Optimization:** *"Refactor this logic to be $O(n)$ or better. Use HashMaps, caching, or more efficient data structures if necessary."*
5. **Benchmark:** Compare the readability and theoretical speed of the new code against the old.

Examples:

1. **The $O(n^2)$ Nested Loop:** *Code:* A script comparing two large arrays by looping one inside the other to find matching user IDs. *Prompt: "This nested loop is an $O(n^2)$ bottleneck. Refactor it to use a JavaScript Set or Map to achieve $O(n)$ time complexity."*
2. **The N+1 Query Problem:** *Code:* Fetching a list of 50 departments, then running a separate database query inside a loop to get the employees for each. *Prompt: "Identify the N+1 query issue here and rewrite this using a single SQL JOIN or Prisma include statement."*
3. **Inefficient File I/O:** *Code:* A Python script reading a massive 5GB log file into memory all at once (file.read()). *Prompt: "This script crashes on large files due to memory exhaustion. Refactor it to process the file line-by-line using a generator."*
4. **Heavy Frontend Re-renders:** *Code:* A React dashboard that re-renders a massive data table every time a user types in an unrelated search bar. *Prompt: "Audit this component for performance. Implement useMemo and useCallback to prevent the data table from re-rendering unnecessarily."*
5. **Blocking the Event Loop:** *Code:* A synchronous, CPU-intensive cryptographic hashing function running on a Node.js main thread. *Prompt: "This synchronous operation is blocking the event loop. Refactor it to use Node.js worker_threads or asynchronous alternatives."*

Simulating a PR Review

Before submitting code to a human team, students should learn to subject their work to a simulated Pull Request (PR) review. By giving Cursor a highly critical persona, students can catch sloppy naming conventions, lack of error handling, and architectural anti-patterns.

Steps:

1. **Finish the Feature:** Have the student complete a block of code they feel proud of.
2. **Open Chat (Cmd+L):** Attach the file using @.
3. **The Roast Prompt:** *"Act as a brutally honest, senior staff engineer reviewing my Pull Request. 'Roast' my code. Point out every single thing I did wrong regarding SOLID principles, clean code, naming conventions, and error handling. Do not hold back."*
4. **Thick Skin:** Read the feedback. It will usually be highly accurate and point out edge cases the student never considered.
5. **Apply the Feedback:** Ask the AI: *"Create a bulleted checklist of actionable fixes based on your review, then help me apply them using Cmd+I."*

Examples:

1. **The "Clean Code" Roast:** *"Review this class using the strict guidelines of Robert C. Martin's 'Clean Code'. Point out any functions that do more than one thing, poor variable names, or unnecessary comments."*
2. **The "Sysadmin Robustness" Roast:** *"Review this bash automation script. Assume the server environment might be missing dependencies, the network might drop, or the target filesystem might be full. Roast my lack of error handling and exit codes."*
3. **The "Enterprise Architecture" Roast:** *"Act as an Enterprise Architect. Review my data pipeline script. Point out where my logic is too tightly coupled and where I should be using dependency injection or interfaces."*
4. **The "Accessibility (a11y)" Roast:** *"Act as a strict UI/UX reviewer. Look at this @LoginForm.html. Point out every WCAG accessibility violation, missing ARIA label, and poor semantic HTML choice I made."*
5. **The "Junior Developer" Roast:** *"Explain the flaws in this code as if I were a junior developer who just submitted their first PR. Be constructive but thoroughly point out my 'code smells' (like duplicated logic or magic numbers)."*

Advanced Prompting – Custom System Prompts

Configuring the "Rules for AI" settings to permanently alter Cursor's personality

By default, Cursor's AI is polite, conversational, and writes code in a generalized style. For professionals, this "chattiness" wastes time. You can permanently alter the AI's personality and output format using the "Rules for AI" setting, ensuring it always writes code that matches your team's specific requirements.

Steps:

1. **Open Settings:** Press Ctrl + Shift + J (or Cmd + Shift + J) to open Cursor Settings.
2. **Navigate to General:** Scroll down to the **"Rules for AI"** section.

3. **Draft the Persona:** Write clear, authoritative instructions about how the AI should behave globally (across all projects).
4. *(Alternative) Project-Specific Rules:* For rules that only apply to one project, create a .cursorrules file in the root directory.
5. **Test the Behavior:** Open a new chat (Cmd + L) and ask a simple coding question to verify the AI applies your new rules.

Examples:

1. **The "No Yapping" Rule:** *"Be extremely terse. Do not apologize. Do not output conversational filler like 'Here is the code.' Only output the requested code, and limit explanations to inline code comments."*
2. **The Functional Programmer:** *"Always prefer functional programming paradigms. Use pure functions, avoid mutable state, and rely on map, filter, and reduce instead of for loops. Never use JavaScript class syntax unless extending a React component."*
3. **The Strict Systems Admin (RHEL/Oracle):** *"You are an expert Red Hat Enterprise Linux (RHEL) and Oracle Linux administrator. Always write shell scripts using strict POSIX compliance. Include set -euo pipefail at the top of every bash script, and prefer dnf over yum."*
4. **The Azure Cloud Architect:** *"When writing infrastructure or cloud logic, default to the latest Azure SDKs. Always implement Managed Identities for authentication instead of hardcoded connection strings."*
5. **The Big Data Analyst:** *"When generating data processing scripts, prioritize memory efficiency. Default to using PySpark or Pandas chunking for large datasets, and always include logging for data transformation steps."*

Using "Pseudo-code" as a prompting language to bridge the gap between thought and syntax.

Natural language (English) is ambiguous; programming languages are strict. Pseudo-code is the perfect bridge. By writing your prompt as structured pseudo-code, you remove the AI's ability to "guess" your business logic, forcing it to follow your exact control flow while it handles the actual syntax.

Steps:

1. **Open Composer (Cmd + I) or Chat (Cmd + L).**
2. **State the Target Language:** Tell the AI what actual language you want the output in (e.g., "Translate this logic into a Python script").
3. **Write the Pseudo-Code:** Use indentation, capitalized keywords (IF, ELSE, FOREACH), and mathematical operators to sketch the logic.
4. **Generate:** Let the AI convert your structural thoughts into perfect syntax.
5. **Review:** Ensure the AI didn't skip any of your defined steps.

Examples:

1. **The Batch Data Pipeline:**

"Write this in Python using Pandas: LOAD large_dataset.csv in chunks FOREACH chunk: IF 'status' is 'ACTIVE': keep row CONVERT 'timestamp' to UTC APPEND to cleaned_data.parquet"

2. **The Microsoft Graph API Sync:**

"Write a Node.js script for this flow: AUTHENTICATE with MSAL (Client Credentials) FETCH users from /v1.0/users FOREACH user: IF user.department == 'Sales': ASSIGN license 'M365_E5' LOG success or error to external file"

3. **The Exponential Backoff Retry:**

"Write a TypeScript wrapper function: TRY executing passed API function CATCH error: IF retries < 3: WAIT (2 ^ retries) * 100ms RETRY function ELSE: THROW 'Max retries reached'"

4. **The File System Cleanup (Cron Job):**

"Write a Bash script for RHEL 9: FIND all files in /var/log/app > 30 days old FOREACH file: COMPRESS file to .tar.gz MOVE compressed file to /mnt/archive/ DELETE original file SEND email summary to admin"

5. **The Localized Payroll Calculator:**

"Write a JavaScript utility function: INPUT: base_salary, days_absent CALCULATE daily_rate = base_salary / 22 CALCULATE deductions = days_absent * daily_rate CALCULATE gross_pay = base_salary - deductions IF gross_pay > 20833: APPLY standard tax brackets for PHP (Philippine Peso) RETURN formatCurrency(net_pay, 'PHP')"

Handling Hallucinations

A "hallucination" is when the AI confidently writes code using a library that doesn't exist, a function that was deprecated years ago, or an API endpoint it just invented.

Steps:

1. **Identify the Suspicion:** If the AI suggests a method you've never heard of, or if the code immediately throws a "Method Not Found" error.
2. **The "Challenge" Prompt:** Open Chat (Cmd + L) and ask the AI to verify its own work: *"Are you sure that method exists in version X?"*
3. **Force Web/Doc Search:** Use @Web or @Docs to force the AI to read the live internet rather than relying on its internal, outdated training data.
4. **Explicit Version Pinning:** Rewrite your initial prompt to strictly enforce the version of the tool you are using.

Examples:

1. **The Version Pin:** *Hallucination:* The AI writes React Router v5 syntax. *Fix Prompt: "I am strictly using React Router v6.* ***Do NOT*** *use useHistory. Rewrite this routing logic using only v6 compatible hooks like useNavigate."*
2. **The Verification Challenge:** *Hallucination:* The AI invents an Azure CLI command. *Fix Prompt: "You suggested az storage blob auto-tier. Does that command actually exist in the current Azure CLI documentation, or did you hallucinate it? Verify this using @Web before answering."*
3. **The Deprecated Library Fix:** *Hallucination:* The AI uses request (a deprecated Node.js library) for HTTP calls. *Fix Prompt: "The request library is deprecated. Look at my @package.json, see that I have axios installed, and rewrite the network call using axios."*
4. **The Microsoft Graph API Reality Check:** *Hallucination:* The AI queries a non-existent endpoint for user data. *Fix Prompt: "You wrote a fetch call to /v1.0/users/{id}/advanced-metrics. That is not a*

standard Graph API endpoint. Using @Docs for Microsoft Graph API, find the correct endpoint to get a user's sign-in activity."

5. **The "Prove It" Prompt:** *Hallucination:* The AI writes a complex regex to parse a Linux configuration file. *Fix Prompt: "I do not trust this Regex. Write a quick unit test with 3 sample lines from a standard sshd_config file, run the test in the terminal, and prove to me that this regex extracts the Port number correctly."*

Future-Proofing & Best Practices

The "Human in the Loop"

While AI makes development faster, it does not remove accountability. The developer is always legally, ethically, and functionally responsible for the code that goes into production.

AI models are statistical prediction engines, not reasoning beings. They do not understand the consequences of the code they write, nor do they possess the institutional knowledge of your specific business rules. If an AI writes a script that drops a production database, the AI will not be fired—the developer will. The "Human in the Loop" (HITL) philosophy ensures the AI acts as an assistant, while the human acts as the editor and final gatekeeper.

Key Insights:

- **The "Looks Good To Me" (LGTM) Trap:** It is incredibly dangerous to accept a massive block of green code in Composer without reading it. AI code is notoriously plausible; it looks correct at a glance even when it contains fatal logical flaws.
- **Contextual Blindness:** The AI only knows what is in its context window. It doesn't know that your company is migrating servers next week or that a specific business partner requires a quirky data format.

Example:

1. **The Business Logic Check:** * *Scenario:* The AI generates a payroll calculation function.
    - *Human Action:* The developer manually reviews the code to ensure it accounts for recent, unannounced company policy changes regarding overtime pay that the AI couldn't possibly know.
2. **The Environment Reality Check:** * *Scenario:* The AI writes a file-handling script utilizing Windows file paths (C:\logs\).
    - *Human Action:* The developer steps in and corrects the paths to Unix standards (/var/log/), knowing this script will actually be deployed on an Oracle Linux or RHEL production server.
3. **The Integration Blindspot:** * *Scenario:* The AI writes a highly efficient data-fetching script for a custom API connector.
    - *Human Action:* The developer reviews the loop and realizes it will trigger API rate limits in the Microsoft 365 ecosystem, and manually instructs the AI to add an exponential backoff strategy.
4. **The Dependency Veto:** * *Scenario:* To solve a date-formatting issue, the AI imports the moment.js library.

- *Human Action:* The developer knows moment.js is bloated and largely deprecated in modern builds, so they reject the code and prompt the AI to use native Intl.DateTimeFormat instead.

5. **The Edge Case Anticipation:** * *Scenario:* The AI writes a flawless function to delete a user account.
    - *Human Action:* The developer pauses the acceptance and asks, *"Wait, what happens to the user's orphaned files when their account is deleted?"* forcing the AI to add a cascade delete mechanism.

Security Risks

When you use cloud-based AI, the code and text in your prompt are transmitted over the internet to be processed by a Large Language Model. Even if a tool claims to have "Privacy Mode" enabled, standard enterprise security policy dictates that you must act as if your prompts are being read by a third party.

Key Insights:

- **Zero-Trust Prompting:** Never paste anything into an AI chat that you wouldn't paste onto a public billboard.
- **Accidental Inclusion:** Cursor's @Codebase feature is powerful, but it can accidentally sweep up .env files or hardcoded credentials if you haven't properly configured your .cursorignore file.

Examples of Prompt Sanitization

1. **Scrubbing Authentication Tokens:**
    - *Dangerous: "Fix this script: fetch('api.bank.com', { headers: { Authorization: 'Bearer xoxb-123456-abcdef' } })"*
    - *Sanitized:* Replace the real token with a placeholder before prompting. *"Fix this script: fetch('api.bank.com', { headers: { Authorization: 'Bearer <INSERT_TOKEN_HERE>' } })"*

2. **Redacting Personally Identifiable Information (PII):**
    - *Dangerous:* Pasting a real CSV of user data (e.g., John Doe, john@email.com, 555-0192) into the chat and asking the AI to write a Python sorting script.
    - *Sanitized:* Use a fake data generator to create a dummy CSV with the exact same column headers, and feed *that* to the AI instead.

3. **Masking Internal Infrastructure:**
    - *Dangerous: "Why is my firewall script failing to connect to our primary database at 192.168.1.105 on port 1433?"*
    - *Sanitized:* Abstract the network topology. *"Why is my firewall script failing to connect to <INTERNAL_DB_IP> on <PORT>?"*

4. **Abstracting Proprietary Algorithms:**
    - *Dangerous:* Pasting your company's highly secretive, proprietary algorithm for predicting stock market trends and asking the AI to optimize it.

- *Sanitized:* Extract only the specific mathematical bottleneck (e.g., a generic matrix multiplication issue) and ask the AI to optimize that isolated mathematical function without providing the financial context.

5. **Managing the .cursorignore File:**
   - *Dangerous:* Allowing the AI to index the entire project folder, including local secrets.json files or unencrypted backup dumps.
   - *Sanitized (Step-by-Step):* 1. Create a .cursorignore file in the root directory. 2. Add *.env, *.pem, secrets.json, and any local database .sqlite files to the list. 3. Verify by typing @secrets.json in the chat to ensure the AI is blocked from reading it.