AI VOID FIELD GUIDE

Step-by-Step Tutorials

Focused, practical tutorials that teach you how to accomplish specific tasks step by step. Learn by doing with clear instructions and working code examples.

AI VOID

2026-06-19 14:07 IST

Contents

CHAPTER 01

Streamline AI Coding with Caveman, Aider, Repomix

> **What you'll build:** An efficient, AI-powered coding workflow for big projects by integrating Caveman for quick replies, Aider for interactive coding, and Repomix for comprehensive repository context, guided by AGENTS.md and handoff.md. **Time needed:** ~60 minutes **Prerequisites:** Basic command-line interface (CLI) knowledge, Python 3.8-3.13 installed (for Aider), Node.js and npm/npx (for Repomix, alternative installs available), Git installed and basic Git knowledge **Version scope:** Aider: official-docs-current; Repomix: official-docs-current; Caveman: official-docs-current **Last verified:** 2026-06-19 against official docs (https://aider.chat/docs/install.html)
>
> ⚡ Note: The "Caveman" component in this workflow refers to a conceptual approach of maintaining a repository of quick, pre-defined AI prompts and replies, rather than a specific installable tool with official documentation. Its integration is about structured prompt management. The installation commands for Aider and Repomix are verified.

The AI Coding Context Problem: Why We Need Better Tools

As developers, we often turn to AI assistants to help us write code, debug, or understand complex systems. While incredibly powerful, LLMs (Large Language Models) have a few common challenges, especially in large, evolving projects:

- **Context Window Limitations:** LLMs have a finite "memory" for the conversation. In big codebases, they quickly "forget" earlier instructions, architectural details, or files they've seen. This leads to repetitive prompting.
- **Inconsistent Replies:** Without structured guidance, AI responses can vary, making it hard to get consistent output for common tasks or questions.
- **Manual Handoffs:** When working on a complex feature, you might interact with different "AI agents" (or simply use the LLM for different roles, like a "feature implementer" then a "code reviewer"). Passing context and instructions between these roles manually is tedious and error-prone.

- **Lack of Repository Awareness:** Standard LLM interactions often lack a deep understanding of the entire project structure, dependencies, and relevant files, leading to less accurate or incomplete suggestions.

These issues force us to constantly re-explain, re-paste code, and manually manage the AI's understanding of our project. This tutorial introduces a workflow and a set of tools designed to tackle these problems head-on, making your AI-assisted development smoother and more efficient.

Caveman: Your AI's Short-Term Memory for Quick Replies

Imagine you frequently ask your AI assistant, "What's the project's main entry point?" or "How do I run the tests?" If you don't provide a clear, concise answer every time, the AI might hallucinate or give a generic response.

What is Caveman (Conceptually)? "Caveman" isn't a single software tool to install. Instead, it represents the practice of maintaining a simple, structured repository (like a `caveman` directory in your project) of pre-defined, concise prompts and their expected AI replies or common instructions. Think of it as a "knowledge base" or "short-term memory" for your AI.

Why Use This Approach?

- **Consistency:** Get the same, correct answer every time for frequently asked questions.
- **Efficiency:** Avoid typing out long, repetitive instructions or context.
- **Context Priming:** Quickly prime your AI with essential project details without consuming valuable context window space with verbose explanations.
- **Reduced Hallucinations:** By providing direct answers, you reduce the AI's tendency to guess or invent information.

Problem it Solves: It solves the problem of AI forgetting basic project information or requiring repetitive, detailed prompts for common, simple queries. It's like giving your AI a cheat sheet for your project.

How it Works (Example): You'd create a file, say `caveman/project_entrypoint.txt`, with the core instruction:

```
The main entry point for this Python project is `src/main.py`.
To run it, use `python src/main.py`.
```

Then, when interacting with your AI, you could simply refer to this file or paste its content for a quick, accurate response, or train your agent to consult such files.

What we accomplished: We've understood the concept of Caveman as a method for managing quick, consistent AI replies, solving the problem of repetitive prompting for basic project info.

Aider: The AI Pair Programmer for Interactive Coding

Aider is a powerful CLI tool that acts as an AI pair programmer, allowing you to interactively code with an LLM directly in your terminal. It can read, write, and modify files in your codebase based on your natural language instructions.

What is Aider? Aider is an open-source command-line interface (CLI) tool that connects to various LLMs (like OpenAI's GPT models) and enables an interactive coding session. You tell Aider what you want to change, and it proposes code modifications, which you can then accept, reject, or refine. It keeps track of your project's files and the changes it makes.

Why Use Aider?

- **Direct File Modification:** Aider can directly edit your project's files, eliminating the need to copy-paste code between your editor and the AI chat interface.
- **Interactive Workflow:** It's designed for a conversational, iterative development process. You propose a task, Aider suggests code, you review, and you refine.
- **Git Integration:** Aider is Git-aware, committing changes as it goes, making it easy to track and revert modifications.
- **Context Management:** It intelligently manages the context it sends to the LLM, focusing on relevant files to stay within token limits.

Problem it Solves: Aider solves the problem of inefficient, fragmented AI coding workflows where developers constantly switch between their IDE and an LLM chat, manually applying suggested changes. It streamlines the coding process by bringing the AI directly into your development environment.

Key Commands and Examples: Once installed, you'll primarily interact with Aider through its command-line interface.

```
# To start an Aider session, you typically specify the files you want it to
work on.
```

```
# For example, to work on 'src/main.py' and 'tests/test_main.py':
aider src/main.py tests/test_main.py
```

Inside the Aider session, you'll use natural language.

```
# Example Aider interaction (what you'd type in the Aider prompt):
# > add a function called 'greet' that takes a name and returns "Hello,
{name}!" in src/main.py
# Aider will then propose changes, which you can accept or modify.
```

What we accomplished: We've learned about Aider's role as an interactive AI pair programmer, understanding how it streamlines coding by directly modifying files and managing context.

Repomix: Mastering Repository Context for LLMs

LLMs struggle with large codebases. Their context window is limited, and simply dumping all files into the prompt is inefficient and expensive. Repomix is designed to solve this.

What is Repomix? Repomix is a powerful CLI tool that processes your entire repository, or specific parts of it, and packages the codebase into various AI-friendly formats. It can intelligently filter files, provide token counts, and structure the output in a way that LLMs can efficiently understand and process.

Why Use Repomix?

- **Context Optimization:** It helps you stay within LLM token limits by providing only the most relevant parts of your codebase.
- **Comprehensive Overview:** Even with token limits, Repomix can give the LLM a much better overview of your project structure, dependencies, and key files than simply listing a few files.
- **AI-Friendly Formats:** It formats the code and metadata in a way that LLMs are trained to understand, improving comprehension and reducing errors.
- **Cost Efficiency:** By optimizing context, Repomix can reduce the number of tokens sent to the LLM, potentially lowering API costs.

Problem it Solves: Repomix solves the critical problem of providing large, complex codebases as context to LLMs. It prevents LLMs from hitting context limits, missing crucial files, or misunderstanding the overall project architecture due to fragmented information.

Key Commands and Examples: Repomix is typically run from the root of your project.

```
# To generate a summary of your repository for an AI:
repomix

# To generate a context file for specific files or directories:
repomix --include src tests

# To see the token count for your repository (useful for LLM context limits):
repomix --tokens
```

The output of `repomix` can then be piped into an LLM, used as input for an AI agent, or simply reviewed to understand the codebase better.

What we accomplished: We've explored Repomix's capabilities in optimizing and formatting repository context for LLMs, addressing a major challenge in AI-assisted development with large projects.

Setting Up Your AI Coding Toolkit (Installation)

Before we dive into the integrated workflow, let's get our tools installed. We'll install Aider and Repomix. Remember that "Caveman" is a conceptual approach to prompt management, not a tool that requires installation.

Prerequisites Check

Ensure you have the following installed:

- **Python 3.8-3.13:** Required for Aider. You can check your version with `python --version`.
- **Node.js and npm/npx:** Required for Repomix. Check with `node --version` and `npm --version`.
- **Git:** Essential for version control and Aider's integration. Check with `git --version`.

1. Install Aider

Aider is a Python package and can be installed using `pip`.

Step 1: Install Aider

Open your terminal and run the following command:

```
pip install aider-chat
```

> ⚡ Note: If you encounter permission issues, you might need to use `pip install --user aider-chat` or activate a virtual environment first. Using a virtual environment is a best practice for Python projects to avoid conflicts.

Step 2: Verify Aider Installation

After installation, you can verify it by checking the version or running a simple help command.

```
aider --version
```

You should see an output indicating the installed Aider version. If you see an error like `command not found`, ensure your Python `Scripts` directory (or equivalent for your OS) is in your system's PATH.

What we accomplished: We've successfully installed and verified Aider, our interactive AI pair programmer.

2. Install Repomix

Repomix is a Node.js package and can be installed globally using `npm` or `npx`. We'll use `npm` for a global installation, which makes it available anywhere on your system.

Step 1: Install Repomix Globally

Open your terminal and run the following command:

```
npm install -g repomix
```

> ⚡ Note: The `-g` flag installs Repomix globally, making it accessible from any directory. If you prefer not to install globally, you can use `npx repomix` to run it directly without installation, but for a consistent workflow, global installation is often more convenient.

Step 2: Verify Repomix Installation

After installation, verify it by checking its version.

```
repomix --version
```

You should see the Repomix version number. If you get a `command not found` error, ensure your `npm` global binaries directory is in your system's PATH.

Alternative: Install Repomix with Homebrew (macOS/Linux)

If you're on macOS or Linux and use Homebrew, you can install Repomix with:

```
brew install repomix
```

Then verify with `repomix --version`.

What we accomplished: We've successfully installed and verified Repomix, our repository context manager.

The Integrated Workflow: AGENTS.md, Handoffs, and AI Tools in Action

Now that our tools are set up, let's put them together into a coherent, AI-powered workflow. This workflow is designed for bigger projects where tasks might be complex, requiring different "AI agents" or roles, and where maintaining context is crucial.

Understanding the Pillars: AGENTS.md and Handoffs

Before diving into the tools, let's understand two critical markdown files that orchestrate our AI workflow:

- **`AGENTS.md`**: Think of this as the "README for AI agents." It's a standardized file in your project's root directory that provides explicit instructions, context, and roles for various AI assistants. It tells your AI how to behave, what its responsibilities are, and what resources it should consult. This prevents repetitive prompting about the AI's role or the project's setup.
    - **Purpose:** Define roles (e.g., "Feature Implementer," "Code Reviewer"), project guidelines, coding standards, preferred technologies, and links to important documentation.
- **`handoff.md`**: When you switch between different AI "roles" or need to pick up a task where another AI (or human) left off, a `handoff.md` file captures the current state, completed tasks, remaining work, and any specific instructions for the next agent.
    - **Purpose:** Ensure seamless transitions, prevent loss of context, and provide clear next steps, especially in a multi-agent or collaborative AI environment.

Mini-Project: Adding a Simple Feature

Let's imagine we have a small Python project and want to add a new feature: a utility function to reverse a string. We'll use our tools to guide an AI through this task.

Step 1: Create a Sample Project Structure

First, create a new directory for our project and initialize a Git repository.

```
mkdir ai_workflow_project
cd ai_workflow_project
git init
```

Now, let's create some basic files: a `src` directory with a `utils.py` file, a `tests` directory with `test_utils.py`, and our `AGENTS.md` file.

```
mkdir src tests
touch src/__init__.py src/utils.py tests/__init__.py tests/test_utils.py AGENT
S.md
```

Step 2: Define Your AI Agent with `AGENTS.md`

We'll create a basic `AGENTS.md` file that defines a "Feature Implementer" agent. This will be the primary context we give to our AI.

Open `AGENTS.md` in your favorite editor and add the following content:

```
# AGENTS.md for AI Workflow Project

This file provides context and instructions for AI agents working on this
project.

## 1. Project Overview

This is a simple Python project containing utility functions.
- **Main language:** Python 3.9+
- **Testing framework:** `pytest`
- **Code style:** PEP 8

## 2. Agent Roles

### 2.1. Feature Implementer

**Role:** Develop new features, implement functions, and write corresponding
unit tests.
**Responsibilities:**
- Write clear, concise, and well-documented Python code.
- Ensure all new code has comprehensive unit tests.
- Adhere to PEP 8 coding standards.
- Update `handoff.md` with progress and next steps after completing a task.
```

```
**Instructions:**
- When implementing a feature, focus on `src/` for logic and `tests/` for
tests.
- Always ask for clarification if instructions are unclear.
- Commit changes frequently and with descriptive messages.

## 3. Important Files

- `src/utils.py`: Contains general utility functions.
- `tests/test_utils.py`: Unit tests for utility functions.
- `AGENTS.md`: This file, for agent instructions.
- `handoff.md`: For task handoffs between agents or humans.
```

Save and close `AGENTS.md`.

What we accomplished: We've set up a basic project and defined a clear role and instructions for our AI using `AGENTS.md`.

Step 3: Use Repomix to Provide Initial Repository Context

Before interacting with Aider, let's use Repomix to generate a comprehensive overview of our (small) project. This is especially valuable in larger projects. We'll output this to a temporary file or directly pipe it to our AI.

```
repomix --include AGENTS.md src tests --output-format markdown > repo_context.
md
```

> ⚡ Note: The `--include` flag tells Repomix to focus on specific files/directories. `--output-format markdown` ensures the output is readable and structured for an LLM. We're redirecting the output to `repo_context.md`. In a real scenario, you might copy-paste this into your LLM's initial prompt or configure your AI agent to read it.

Now, inspect `repo_context.md`. You'll see a structured representation of your project, including file paths and content. This is what an LLM would consume for a deep understanding.

What we accomplished: We've used Repomix to generate an optimized, AI-friendly context file for our project, preparing the ground for our AI assistant.

Step 4: Use Caveman (Conceptually) for Quick Replies

While we don't have a specific "Caveman" tool to run, we can simulate its use. Imagine you have a directory `caveman_prompts` with common queries.

```
mkdir caveman_prompts
echo "Please explain the purpose of the 'src/utils.py' file in this project."
> caveman_prompts/explain_utils.txt
```

When interacting with Aider, if you needed to quickly remind it about the purpose of `src/utils.py`, you could conceptually "use" this prompt. For now, we'll just keep it in mind as a pattern.

What we accomplished: We've conceptually prepared a "Caveman" prompt, demonstrating how pre-defined quick replies can be managed for efficiency.

Step 5: Implement Feature with Aider

Now, let's use Aider to implement our string reversal function and its test. We'll start Aider, giving it `AGENTS.md`, `src/utils.py`, and `tests/test_utils.py` as initial context.

```
aider AGENTS.md src/utils.py tests/test_utils.py
```

Aider will start an interactive session. It will likely read `AGENTS.md` first.

Aider Interaction (What you would type in the Aider prompt):

```
> Please implement a function `reverse_string(s: str) -> str` in `src/
utils.py` that reverses the input string.
> Also, add a corresponding unit test in `tests/test_utils.py` using `pytest`
to ensure it works correctly for a few cases, including an empty string.
```

Aider will process your request and propose changes. It might look something like this:

```
# Aider will show you a diff. Review it carefully.
# For example:
#
# --- src/utils.py
# +++ src/utils.py
# @@ -0,0 +1,4 @@
# +def reverse_string(s: str) -> str:
# +    return s[::-1]
# +
# --- tests/test_utils.py
# +++ tests/test_utils.py
# @@ -0,0 +1,7 @@
# +from src.utils import reverse_string
# +
# +def test_reverse_string():
# +    assert reverse_string("hello") == "olleh"
# +    assert reverse_string("Python") == "nohtyP"
# +    assert reverse_string("") == ""
```

```
#
# Accept changes? (y/n/diff/quit)
```

Type `y` to accept the changes. Aider will apply them and commit them to Git.

Step 6: Verify the Feature

After Aider commits, you can exit Aider by typing `/quit`. Then, let's manually run the tests to ensure everything works as expected.

```
# First, install pytest if you haven't already
pip install pytest

# Now, run the tests
pytest tests/test_utils.py
```

You should see output indicating that the tests passed.

```
============================= test session starts
==============================
platform ... -- Python 3.9.x
plugins: ...
collected 1 item

tests/test_utils.py .
[100%]

============================== 1 passed in ...s
===============================
```

What we accomplished: We successfully used Aider, guided by `AGENTS.md`, to implement a new feature and its unit tests, and then verified its correctness.

Step 7: Create a `handoff.md` (Conceptual)

Imagine now that the "Feature Implementer" agent has completed its task, and you want to pass this work to a "Code Reviewer" agent or a human colleague. A `handoff.md` file would capture the state.

Create a `handoff.md` file:

```
# Handoff Report: String Reversal Feature

## 1. Task Completed

- Implemented `reverse_string(s: str)` function in `src/utils.py`.
- Added unit tests in `tests/test_utils.py`.
- All tests pass locally.

## 2. Next Steps / Reviewer Instructions
```

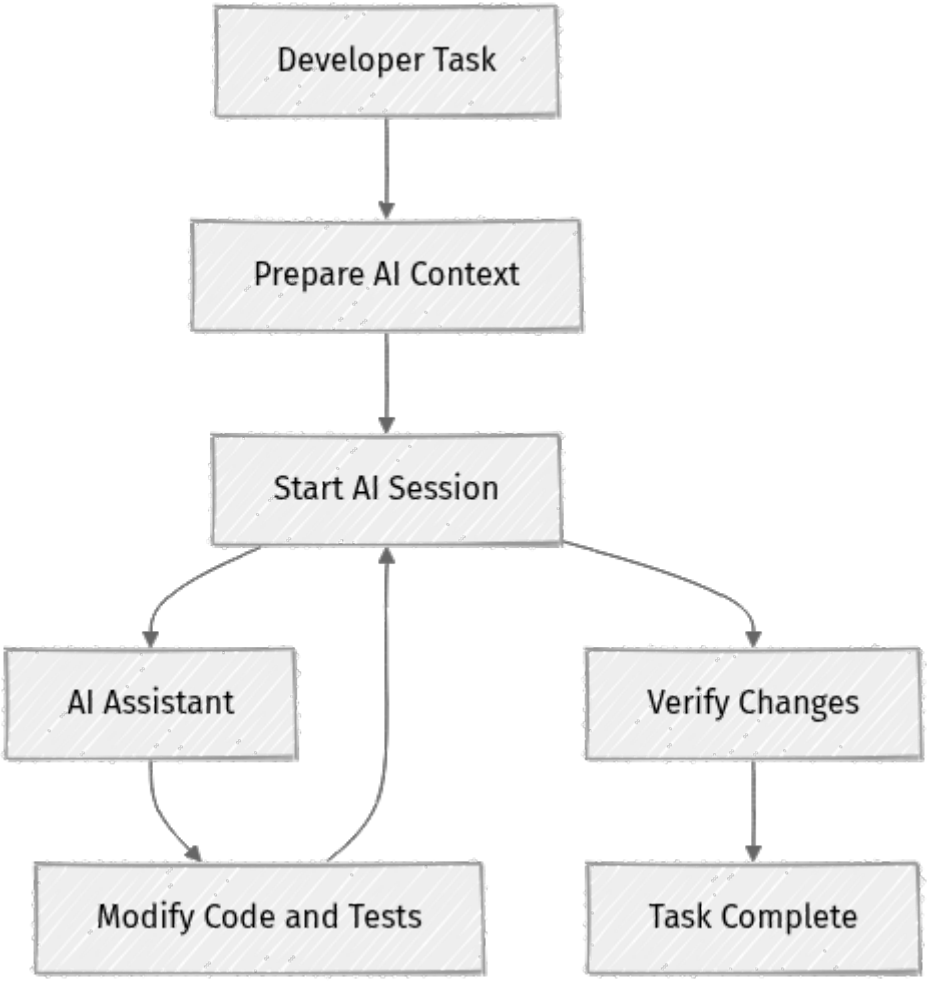
```
**For the Code Reviewer Agent:**
- Please review `src/utils.py` for code quality, adherence to PEP 8, and
potential edge cases.
- Review `tests/test_utils.py` for test coverage and effectiveness.
- Ensure proper type hints are used.
- Provide feedback in a new `review_feedback.md` file.

## 3. Relevant Commits

- (Aider automatically commits, you would list the commit hash here or point
to `git log`)

## 4. Context Provided

- Initial context was provided via `AGENTS.md` and `repo_context.md`.
```

This `handoff.md` would then be part of the context for the next AI agent or human taking over.

What we accomplished: We've created a `handoff.md` to demonstrate how context can be passed between different stages or agents in a structured manner.

Integrated Workflow Diagram

Here's a simple flow of how these components work together:



What we accomplished in this section: We've walked through a complete, integrated AI coding workflow, leveraging `AGENTS.md` for role definition, Repomix for comprehensive context, Caveman (conceptually) for quick replies, and Aider for interactive code implementation and testing. We also learned how `handoff.md` facilitates seamless transitions.

Common Pitfalls and Best Practices for AI-Assisted Development

While powerful, AI tools require careful handling. Here are some common mistakes and best practices to keep your workflow efficient and reliable.

Common Pitfalls

- **Vague Prompts:** Asking "Make this better" without specific criteria leads to generic, unhelpful, or even detrimental changes.

  > ⚠️ Common mistake: "Fix this code."

- **Skipping Local Testing:** Over-reliance on AI to produce perfect code without local verification is a recipe for bugs. AI can introduce subtle errors or misunderstand requirements.

  > ⚠️ Common mistake: Assuming AI-generated code is always correct and skipping `pytest` or manual checks.

- **Ignoring Context Limits:** In large projects, dumping too many files into an LLM will either hit token limits, incur high costs, or dilute the AI's focus.

  > ⚠️ Common mistake: Not using tools like Repomix and just feeding an AI many irrelevant files.

- **Lack of Structured Communication:** Without `AGENTS.md` or `handoff.md`, you'll spend valuable time re-explaining project setup, coding standards, or previous work.

  > ⚠️ Common mistake: Treating every AI interaction as a fresh start, without building on previous context or defined roles.

- **Over-reliance on AI for Design:** AI excels at implementation and boilerplate, but complex architectural decisions or novel solutions still benefit greatly from human insight.

> ⚠️ Common mistake: Asking AI to design an entire system from scratch without providing high-level constraints or principles.

Best Practices

- **Be Specific and Iterative:** Break down complex tasks into smaller, manageable steps. Provide clear, detailed instructions. Review AI output after each step and refine your prompts.

> ⚡ Pro tip: "Refactor `function_x` to improve readability by using `list comprehensions` and adding `type hints`."

- **Always Verify Locally:** Treat AI-generated code as a first draft. Run tests, linting, and manually review the code.

> ⚡ Real-world insight: In production, AI is a co-pilot, not the pilot. Human oversight is paramount for quality and security.

- **Manage Context Actively with Repomix:** Use Repomix to intelligently select and format the most relevant parts of your codebase for the AI. This keeps context windows focused and costs down.

> ⚡ Optimization: Experiment with Repomix's `--include`, `--exclude`, and `--output-format` options to find the optimal context for your LLM.

- **Define Roles with `AGENTS.md`:** Clearly define the AI's role, responsibilities, and constraints in an `AGENTS.md` file. This acts as a consistent guide for the AI, especially across multiple sessions or tasks.

> 📌 Key Idea: `AGENTS.md` is your AI's job description and rulebook.

- **Use Handoffs for Continuity:** For multi-step tasks or when switching AI "agents" (or even human collaborators), use `handoff.md` to summarize progress and outline next steps.

- **Leverage "Caveman" for Boilerplate:** For frequently asked questions or common code snippets, maintain a "Caveman" repository of quick replies. This ensures consistency and saves time.
- **Understand AI Limitations:** Recognize that LLMs can hallucinate, produce suboptimal code, or misinterpret nuances. Always apply critical thinking.
- **Version Control is Your Friend:** Aider integrates with Git, but always be prepared to review and revert changes. Use Git's branching features for experimental AI-assisted development.

What we accomplished: We've identified common pitfalls in AI-assisted development and established best practices for a more effective, reliable, and efficient workflow using our integrated toolkit.

Next Steps: Expanding Your AI Coding Workflow

You've now learned how to integrate Caveman (concept), Aider, and Repomix into a powerful AI coding workflow. But this is just the beginning! Here are three concrete ideas to expand and refine your new system:

1. **Automate Context Provisioning:** Instead of manually piping `repo_context.md` or `AGENTS.md` into your AI, explore ways to automate this.
    - **Challenge:** Can you write a small `bash` script that first runs `repomix`, then launches `aider` with the `AGENTS.md` and the `repo_context.md` content already loaded into the prompt?
    - **Idea:** Many AI platforms allow you to define system prompts or initial messages. You could configure your preferred LLM client to automatically prepend the content of `AGENTS.md` and a concise `repomix` output.
2. **Develop Advanced "Caveman" Prompts:** Expand your `caveman_prompts` directory with more sophisticated, multi-line instructions for common development tasks.
    - **Challenge:** Create "Caveman" prompts for tasks like "generate a Dockerfile for this project," "explain the project's dependency tree," or "write a simple README.md based on project files."
    - **Idea:** Categorize your prompts (e.g., `caveman/refactoring/`, `caveman/testing/`). You could even build a small script to quickly search and insert these prompts into your Aider session or LLM chat.

3. **Integrate with CI/CD for AI-Driven Quality Checks:** Explore how Repomix's output could be used in a CI/CD pipeline.
    - **Challenge:** Imagine a pull request. Can Repomix generate a summary of the changes in that PR, and then an LLM (perhaps via a custom script) use that summary to perform an automated code review against the `AGENTS.md` standards?
    - **Idea:** Use Repomix to create a concise "diff context" for new pull requests. A custom bot could then feed this context, along with `AGENTS.md` rules, to an LLM to generate an initial automated review comment, focusing on specific quality gates defined in your `AGENTS.md`.

By taking these next steps, you'll move beyond basic AI assistance to a truly integrated, intelligent development environment that adapts to your project's unique needs. Happy coding!