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

2026-05-05 19:03 IST

Contents

01 Master Pi Coding Agent Workflows 3

CHAPTER 01

Master Pi Coding Agent Workflows

> **What you'll build:** By the end of this tutorial, the reader will be able to proficiently use Pi Coding Agent to streamline daily software engineering tasks, develop custom agentic workflows, and integrate with advanced AI models and tools. **Time needed:** ~240 minutes **Prerequisites:** Basic command-line interface (CLI) proficiency, Fundamental software development concepts, Node.js and npm installed, An AI model provider API key (e.g., OpenAI) **Version used:** unknown

---

Introduction to Pi Coding Agent and Agentic Workflows

Welcome to the world of agentic coding, where your terminal becomes a powerful co-pilot, not just a command input. In this tutorial, we're going to dive deep into Pi Coding Agent, a truly minimal yet incredibly extensible tool designed to revolutionize your daily software engineering tasks.

What is Pi Coding Agent?

Pi is a command-line interface (CLI) coding agent that acts as a smart assistant directly within your terminal. It's built with a "small core, extensible periphery" philosophy, meaning its fundamental operations are lean, but its capabilities can be vastly expanded through custom extensions, skills, and prompt templates written in TypeScript (and Python for some integrations). Think of it as a highly adaptable, AI-powered shell that understands context, executes commands, and helps you navigate complex coding problems.

Why Agentic Workflows Matter

Modern software development often involves repetitive tasks, context switching, and complex problem-solving. Agentic workflows, powered by tools like Pi, aim to automate these burdens. Instead of manually searching documentation, writing boilerplate code, or debugging line by line, you can delegate these tasks to an intelligent agent. This frees up your cognitive load, allowing you to focus on higher-level design and architectural challenges.

> 📌 **Key Idea:** Pi brings the power of AI directly into your terminal, enabling intelligent automation and collaboration on coding tasks.

The Problem Pi Solves

Pi addresses the challenge of integrating AI assistance seamlessly into a developer's existing workflow. Traditional AI coding assistants often operate in separate UIs or require extensive context copying. Pi, by living in your terminal, works directly within your project directory, understands your file system, and can execute commands, making it an active participant in your development process rather than just a suggestion engine.

Real-World Impact

Imagine asking your terminal to "implement a new API endpoint for user authentication" or "find and fix all `null` pointer exceptions in this module." Pi can interpret these requests, execute relevant commands, modify code, and even explain its reasoning, making complex tasks feel significantly simpler and faster. This translates to reduced development cycles and improved code quality.

> ⚡ **Real-world insight:** Companies are increasingly adopting agentic tools to accelerate feature development and improve code quality, especially in areas like refactoring, test generation, and incident response.

What Can Go Wrong: Misconceptions

A common pitfall with agentic tools is expecting them to be fully autonomous from day one. Pi is a powerful assistant, not a replacement for human developers. It excels when given clear instructions and when you're prepared to review and guide its actions. Misunderstanding this can lead to frustration if the agent doesn't immediately grasp complex, ambiguous tasks without iterative prompting.

By the end of this section, you should have a clear understanding of what Pi Coding Agent is, why agentic workflows are crucial for modern development, and the core problem Pi aims to solve within your terminal environment.

Installation and Initial Configuration (Providers, Local/Cloud Models)

Now that you understand the "why," let's get hands-on and set up Pi Coding Agent on your system. This section will guide you through the installation process, initial configuration, and how to connect Pi to various AI models, both cloud-based and potentially local.

Step 1: Install Node.js and npm

Pi Coding Agent is built on Node.js and distributed via npm. If you don't already have them, you'll need to install them first.

You can download Node.js (which includes npm) from the official website: https://nodejs.org/. Follow the instructions for your operating system.

To verify your installation, open your terminal and run:

```
node -v
npm -v
```

You should see version numbers for both `node` and `npm`.

Step 2: Install Pi Coding Agent

With Node.js and npm ready, installing Pi is a single command. We'll install it globally so you can run `pi` from any directory.

```
npm install -g @mariozechner/pi-coding-agent
```

This command downloads and installs the `pi-coding-agent` package globally on your system.

> ⚠️ **Common mistake:** If you encounter permission errors during global installation (e.g., `EACCES`), you might need to fix npm permissions or use `sudo` (though fixing permissions is generally preferred). A common fix involves changing the ownership of npm's directories. Search for "npm EACCES fix" for detailed instructions.

Once the installation completes, verify it by running:

```
pi --version
```

You should see the installed version of Pi Coding Agent.

Step 3: Configure Your AI Model Provider

Pi needs access to an AI model to function. We'll start with OpenAI, a popular choice. You'll need an OpenAI API key. If you don't have one, sign up at platform.openai.com and generate a new secret key.

To make your API key available to Pi, set it as an environment variable. This is a secure way to handle sensitive credentials.

For Linux/macOS:

```
export OPENAI_API_KEY="sk-YOUR_OPENAI_API_KEY_HERE"
```

For Windows (Command Prompt):

```
set OPENAI_API_KEY="sk-YOUR_OPENAI_API_KEY_HERE"
```

For Windows (PowerShell):

```
$env:OPENAI_API_KEY="sk-YOUR_OPENAI_API_KEY_HERE"
```

> ⚡ **Note:** Replace `"sk-YOUR_OPENAI_API_KEY_HERE"` with your actual OpenAI API key. For persistent access, you should add this `export` or `set` command to your shell's configuration file (e.g., `.bashrc`, `.zshrc`, `config.fish`, or Windows system environment variables).

Step 4: First Pi Session and Model Selection

Now you're ready to start your first Pi session! Navigate to a project directory where you want Pi to work, or just start it in an empty directory to experiment.

```
cd /path/to/your/project # Or any directory
pi
```

When you run `pi` for the first time, it might ask you to confirm some initial setup or just present you with a prompt.

You can explicitly specify the provider and model you want to use. For example, to use OpenAI's `gpt-4o` model:

```
pi --provider openai --model gpt-4o
```

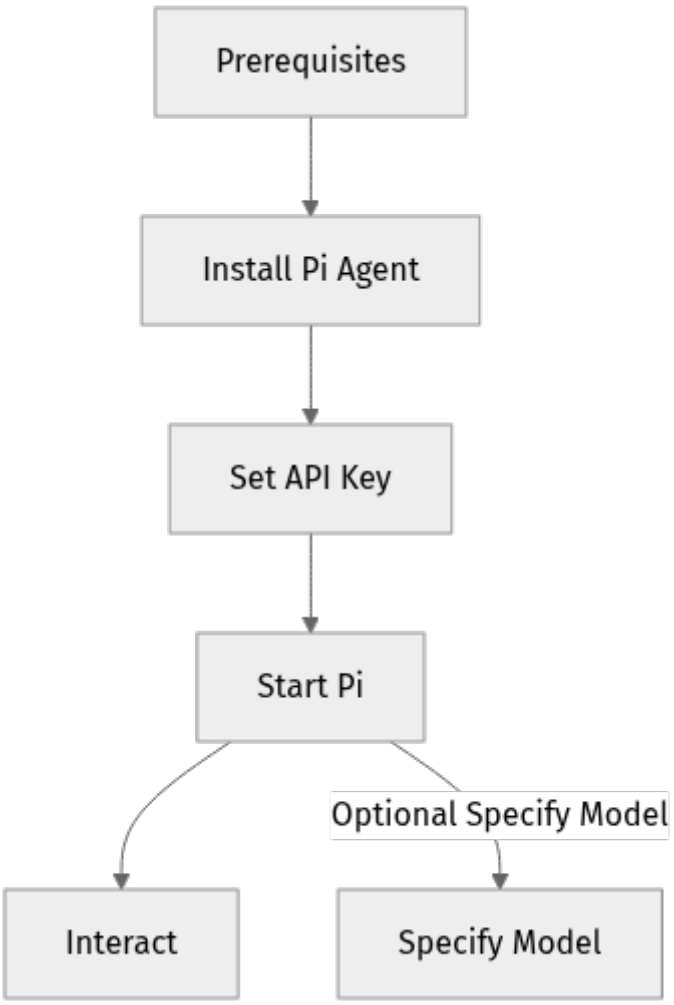
> ⚡ **Quick Note:** You can list available models by running `pi --list-models`. This will show you which providers and models Pi can connect to based on your configuration.

Step 5: Understanding Local vs. Cloud Models

Pi is flexible enough to work with both cloud-based AI models (like OpenAI, Anthropic, etc.) and models running locally on your machine.

- **Cloud Models:** These are powerful, generally faster, and require less local compute. They involve API calls and associated costs. Configuration typically involves setting API keys.
- **Local Models:** These run entirely on your hardware, offering privacy and no per-token costs. They are ideal for sensitive code or when you want full control. To use local models, you typically need a local inference server (like Ollama) running, which Pi can then connect to as a custom provider.

To use a local model, you would generally: 1. Install a local inference server (e.g., Ollama). 2. Download the desired model (e.g., `llama3`). 3. Configure Pi to point to the local server's API endpoint, potentially defining a custom provider in Pi's configuration files (which we'll explore later).



This diagram illustrates the basic flow from prerequisites to your first interaction with Pi.

By completing this section, you have successfully installed Pi Coding Agent, configured it with an AI model provider, and initiated your first agentic session. You're now ready to start interacting with your AI coding assistant!

Mastering Terminal Workflows (Sessions, Branching, Model Switching)

Pi is designed to be a fluid, interactive experience within your terminal. This section will teach you how to effectively manage your conversations with Pi, explore different solutions, and switch models on the fly to optimize your workflow.

The Power of Pi Sessions

When you run `pi`, you start a session. This session maintains context, meaning Pi remembers your previous prompts and its own responses. This allows for natural, conversational problem-solving.

```
# Start a new session in your project directory
cd my-awesome-project
pi
```

Once inside a session, you can type your requests directly. For example:

```
> pi
pi: How can I help you?
user: Create a simple Node.js script that fetches data from a public API and
logs it.
pi: Okay, which public API would you like to use?
user: Let's use the JSONPlaceholder /todos endpoint.
# ... Pi will then generate code and explanations ...
```

Rewinding and Branching: Exploring Alternatives

One of Pi's most powerful features is its ability to "rewind" to a previous point in your conversation and "branch" off from there. This is incredibly useful for: * Trying a different approach without losing the original context. * Experimenting with different prompts. * Debugging by stepping back to a specific interaction.

Imagine you've asked Pi to write some code, and it didn't quite hit the mark. Instead of starting over, you can rewind.

Let's say your conversation looked like this: 1. User: "Write a function to sum numbers." 2. Pi: (provides a basic `for` loop sum) 3. User: "Can you make it recursive?" 4. Pi: (provides a recursive sum function) 5. User: "Actually, I needed an iterative approach, but using `reduce`."

At step 5, you realize you want to go back to step 2's context and ask for `reduce`.

To rewind, you'll typically use a command like `pi --rewind N` or an interactive rewind mechanism within the Pi UI (if available, check `pi --help` for specifics). The core idea is to jump back to a specific message ID or position in the history.

Once you've rewound, any new prompt you give will create a new "branch" of the conversation, leaving the old path intact. This is like version control for your AI interactions!

> ⚡ **Note:** The exact command for rewinding and branching might vary slightly with Pi updates. Always refer to `pi --help` or the official documentation (`pi.dev/docs/latest`) for the most current syntax. The conceptual flow remains the same.

Dynamic Model Switching

Different AI models excel at different tasks. `gpt-4o` might be great for complex reasoning, while a smaller, faster model might be better for quick code generation. Pi allows you to switch models dynamically, even within a session.

To switch models, you can often use `pi --model <new_model>` or a similar command, or even configure a specific model for a new branch.

For example, if you're in a session and want to try a different model:

```
# Assuming you're in a Pi session
user: I need to refactor this complex function. Let's try it with gpt-4o.
pi: (acknowledges model switch, or you can exit and restart with the flag)
```

To explicitly start a new branch with a different model:

```
# Exit current pi session (Ctrl+D or :q)
# Then start a new pi session in the same directory, specifying the model
pi --model gpt-4o --branch new-refactor-attempt
```

> 🧠 **Important:** `pi --list-models` is your best friend here. It shows you all available providers and models, helping you choose the right tool for the job. You might need to configure additional providers (e.g., Anthropic, custom local models) in Pi's settings for them to appear in this list.

Sharing Sessions

Pi sessions can often be shared, allowing you to collaborate or review past interactions. The official `pi.dev` site mentions being able to "share the entire session." This typically involves exporting the session history to a file or a shareable link, depending on Pi's features. This is invaluable for code reviews, pair programming, or creating documentation of your development process.

By mastering sessions, branching, and model switching, you gain fine-grained control over your agentic workflows, allowing you to experiment efficiently and leverage the best AI models for each specific task.

Customizing Pi with Extensions, Skills, and Prompt Templates

Pi's true power lies in its extensibility. It's designed to be a minimal core that you can tailor precisely to your development needs. This section will guide you through creating custom extensions, defining skills, and leveraging prompt templates to make Pi an indispensable part of your workflow.

The Extensible Core: TypeScript Extensions

Pi allows you to write custom extensions using TypeScript. These extensions can define new commands, integrate with external tools, or implement complex logic that Pi can then execute. Think of them as plugins that enhance Pi's capabilities beyond its default set.

Let's create a simple "hello world" extension.

Step 1: Create an Extension Directory

Pi typically looks for extensions in specific directories, often a `.pi` folder in your home directory or project root. Let's assume a project-level extension for now.

```
# In your project directory
mkdir -p .pi/extensions
cd .pi/extensions
```

Step 2: Initialize a TypeScript Project

You'll need a `package.json` and a `tsconfig.json` for your TypeScript extension.

```
npm init -y
npm install --save-dev typescript @types/node
npx tsc --init
```

Now, open `tsconfig.json` and ensure `outDir` is set (e.g., `"outDir": "./dist"`) and `rootDir` is set (e.g., `"rootDir": "./src"`). Also, ensure `target` is set to a modern version like `es2020` or `es2022`.

Step 3: Write Your First Extension

Create a `src` directory and an `index.ts` file inside it:

```
mkdir src
touch src/index.ts
```

Now, open `src/index.ts` and add the following code:

```
// src/index.ts
import { Extension, Agent } from '@mariozechner/pi-coding-agent/dist/
types'; // Adjust path if needed

// Define your custom command
class MyCustomCommandExtension implements Extension {
  id = 'my-custom-command';
  name = 'My Custom Command Extension';
  description = 'A simple extension with a custom command.';

  async activate(agent: Agent): Promise<void> {
    agent.addCommand('hello-pi', async (args: string[]) => {
      const name = args[0] || 'World';
      agent.say(`Hello, ${name} from your custom Pi extension!`);
    });

    agent.say('My Custom Command Extension activated!');
  }

  async deactivate(agent: Agent): Promise<void> {
    agent.say('My Custom Command Extension deactivated.');
  }
}

export default MyCustomCommandExtension;
```

> ⚡ **Note:** The exact `import` path for `Extension` and `Agent` types might vary slightly depending on the Pi version. You might need to inspect the installed `node_modules/@mariozechner/pi-coding-agent` directory to find the correct path (e.g., `dist/types` or `index.d.ts`).

Step 4: Compile Your Extension

Compile your TypeScript code:

```
npx tsc
```

This will create a `dist` directory with your compiled JavaScript.

Step 5: Load and Use Your Extension

Pi needs to know about your extension. You might need to configure Pi's `config.ts` or `config.json` (check `pi.dev/docs/latest/settings`) to point to your extension's `dist` folder. Alternatively, Pi might automatically discover extensions in a `.pi/extensions` directory.

Once loaded, you can ask Pi to use your command:

```
> pi
pi: How can I help you?
user: Use the hello-pi command.
pi: Hello, World from your custom Pi extension!
user: Use the hello-pi command with argument "Instructor".
pi: Hello, Instructor from your custom Pi extension!
```

This demonstrates how to extend Pi's functionality with custom logic.

Defining Custom Skills

Skills are often a more abstract way to describe capabilities that Pi can leverage. While extensions provide the implementation, skills describe what Pi can do. You might define a "code-review" skill or a "test-generation" skill. Pi's core then uses its reasoning capabilities to match your request to an available skill and execute the underlying extension or command.

Skills can sometimes be defined through simple configuration files (e.g., Markdown or YAML) that describe the skill's purpose and how to invoke it.

Prompt Templates for Consistent Workflows

Prompt templates are pre-defined structures for your prompts, ensuring consistency and guiding Pi towards specific outputs. They are often Markdown files (`.md`) that Pi can reference.

Step 1: Create a Prompt Template File

Let's create a template for generating commit messages.

```
# In your project directory
mkdir -p .pi/prompts
touch .pi/prompts/commit-message.md
```

Step 2: Add Content to the Template

Open `commit-message.md` and add:

```
You are an expert software engineer.
Generate a concise and descriptive Git commit message following Conventional
Commits specification.
The message should include a type (feat, fix, docs, style, refactor, test,
chore, perf, build), an optional scope, and a subject.
The body should explain the "what" and "why" of the changes.

Based on the following diff or changes:
---
{{diff}}
---

Commit message:
```

Step 3: Use the Prompt Template

Now, when you interact with Pi, you can instruct it to use this template. You'd typically feed it the `diff` content.

```
# First, generate a diff (e.g., for staged changes)
git diff --staged > changes.diff

# Then, ask Pi to use the template, injecting the diff
pi --template .pi/prompts/commit-message.md --var diff="$(cat changes.diff)"
```

Pi will then use the content of `changes.diff` to fill the `{{diff}}` placeholder in your template and generate a commit message.

> ⚡ **Real-world insight:** Prompt templates are crucial for creating repeatable, high-quality outputs from AI agents. They reduce "prompt engineering" overhead in daily use and ensure the agent adheres to specific standards (like Conventional Commits).

By mastering extensions, skills, and prompt templates, you transform Pi from a general-purpose assistant into a highly specialized, integrated tool that understands and automates your unique development processes.

Building Advanced Agentic Workflows (Self-Extending, Multi-Agent)

Pi's architecture allows for sophisticated agentic behaviors, including the ability for the agent to extend its own capabilities and collaborate with other specialized agents. This section explores these advanced concepts to unlock Pi's full potential.

Self-Extending Agents

One of the most powerful features of Pi is its ability to be "self-extending." This means you can actually ask Pi to write, debug, and implement its own extensions or modify its own configuration. The `pi.dev` website highlights this by stating, "Here we ask Pi to build a custom workflow extension with custom TUI to streamline the commit and push process."

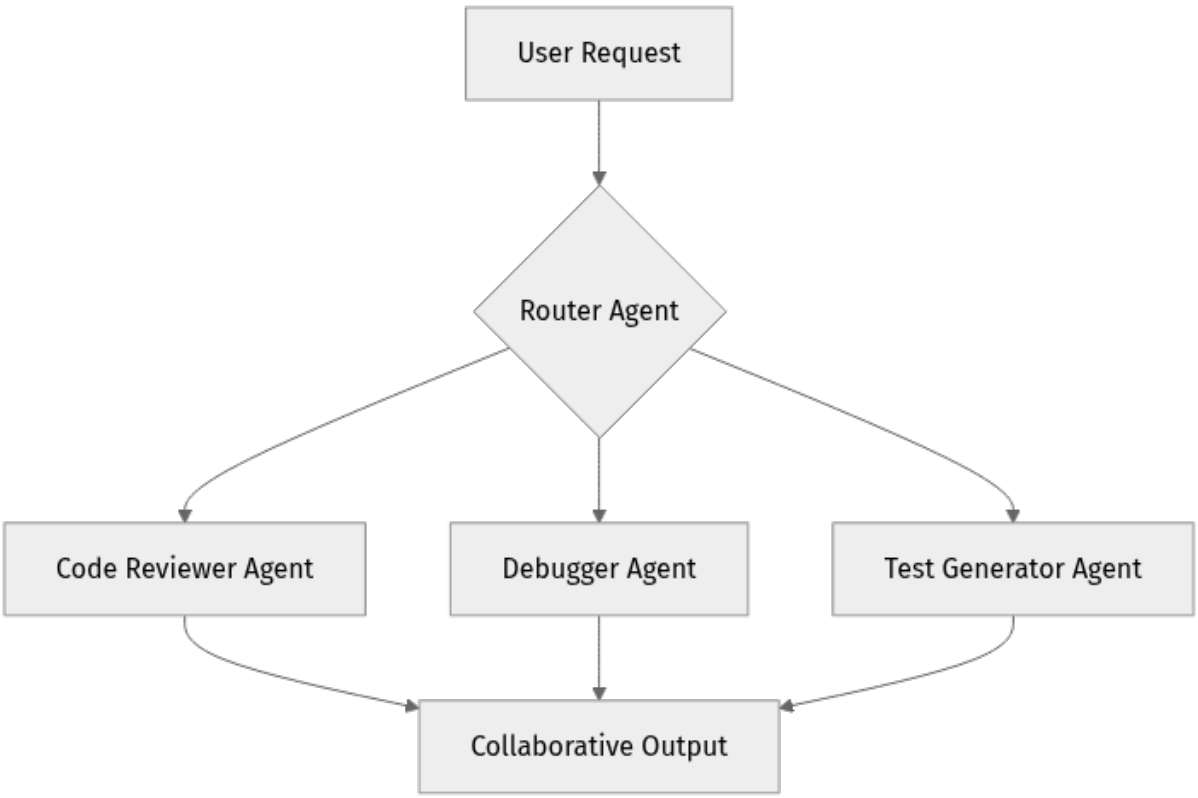
How it works: 1. **Prompt:** You describe a new feature or workflow you want Pi to have. For example, "Create an extension that automatically runs `npm test` after I modify a `.ts` file." 2. **Reasoning:** Pi analyzes your request, understands the context (file changes, `npm test`, TypeScript), and determines that an extension is the appropriate solution. 3. **Code Generation:** Pi generates the TypeScript code for the extension, including file watching logic and command execution. 4. **Integration:** Pi can then guide you on where to save the file, how to compile it, and how to load it into its own configuration. In advanced scenarios, it might even automate these steps. 5. **Testing/Debugging:** You can then ask Pi to help test the new extension and debug any issues that arise.

> 🔥 **Optimization / Pro tip:** When asking Pi to self-extend, break down complex requests into smaller, manageable steps. Start with the core functionality, then add error handling, configuration options, and UI elements incrementally. This mimics how you'd develop software yourself.

Multi-Agent Workflows with `AGENTS.md`

For more complex problems, a single agent might not be sufficient. Multi-agent workflows involve multiple specialized agents collaborating to achieve a larger goal. Pi supports this by allowing you to define different "specialists" or roles. The search evidence mentions a Medium article where someone "turned Pi into a multi-agent coding assistant" by adding specialists via `.md` files and referencing them in config. This suggests a declarative way to define agents.

Conceptual Flow of a Multi-Agent System:



In this diagram: * A **Router Agent** (or a primary Pi instance with routing logic) takes the initial request. * It delegates sub-tasks to specialized agents (Code Reviewer, Debugger, Test Generator). * These agents work in parallel or sequentially, contributing to a **Collaborative Output**.

Implementing with `AGENTS.md` (or similar configuration):

You can define different agent roles and their specific instructions in a Markdown file, perhaps named `AGENTS.md` in your project root or Pi's configuration directory.

`AGENTS.md` Example:

```
# Agent Definitions

## Agent: Code Reviewer
**Role:** An expert in code quality, best practices, and security.
**Instructions:**
- Analyze provided code for potential bugs, performance issues, and adherence
to coding standards.
- Suggest improvements and provide concrete code examples.
- Focus on readability and maintainability.
**Model Preference:** gpt-4o

## Agent: Test Generator
**Role:** A skilled test engineer.
**Instructions:**
- Generate comprehensive unit tests (e.g., Jest, Pytest) for given code.
- Ensure high test coverage and edge case handling.
- Provide examples of test data.
**Model Preference:** gpt-3.5-turbo

## Agent: Debugger
**Role:** An experienced debugger, proficient in identifying and resolving
runtime errors.
**Instructions:**
- Analyze error messages and stack traces.
- Pinpoint the root cause of issues.
- Suggest potential fixes and debugging strategies.
**Model Preference:** gpt-4
```

How Pi uses this: You would then interact with Pi, and depending on your prompt, Pi (or a custom extension you build) could: 1. **Identify Intent:** Determine if the task requires a specialist (e.g., "Review this code for issues" -> Code Reviewer). 2. **Delegate:** Pass the relevant code/context to the designated specialist agent (which might be another Pi instance running with different prompt templates or a specific model). 3. **Synthesize:** Collect the outputs from the specialist agents and present a consolidated response.

> ⚡ **Real-world insight:** Multi-agent systems are particularly effective for complex tasks that require diverse expertise, such as full-stack feature development, security audits, or comprehensive code refactoring, where different agents handle front-end, back-end, database, and testing concerns.

By leveraging self-extension and multi-agent patterns, you can build incredibly powerful and automated workflows that tackle complex software engineering challenges with unprecedented efficiency.

Integrating Pi with External Tools (OpenClaw, Local AI)

Pi's utility extends beyond its core terminal interface. It's designed to be a flexible component within a broader AI ecosystem, capable of integrating with other powerful tools and leveraging local AI inference.

Pi and OpenClaw: A Minimal Agent in a Larger Framework

The search evidence mentions Pi as "The Minimal Agent Within OpenClaw." This suggests a powerful relationship:

- **OpenClaw:** Likely a larger, more comprehensive agentic framework or platform designed for broader automation and control. It might provide advanced orchestration, communication channels, and tool access for multiple agents.
- **Pi:** Acts as a focused, minimal agent within this larger OpenClaw ecosystem. It specializes in terminal-based coding tasks, file system interaction, and code generation.

Think of it like this: OpenClaw provides the "brain" and "nervous system" for complex, multi-step operations, while Pi provides the "hands" and "eyes" for deep, interactive coding within the terminal.

Integration Benefits:

- **Enhanced Automation:** OpenClaw could orchestrate Pi to perform specific coding tasks as part of a larger automated workflow (e.g., "Deploy new feature -> Ask Pi to generate API docs -> Update CI/CD pipeline").
- **Wider Tool Access:** OpenClaw might grant Pi access to tools beyond its native capabilities, such as web browsers (for research), external databases, or cloud services.
- **Multi-Agent Orchestration:** OpenClaw could manage multiple Pi instances or other types of agents, assigning them different roles in a complex development pipeline.

> 🧠 **Important:** While Pi can operate standalone, its integration with frameworks like OpenClaw signifies a trend towards interconnected AI agents working collaboratively across different environments and tasks.

Leveraging Local AI Models

Using local AI models with Pi offers significant advantages, including data privacy, reduced costs (no API charges), and the ability to work offline. While cloud models are powerful, local inference provides control and can be tailored to specific needs.

How Pi integrates with Local AI:

1. **Local Inference Server:** You'll typically run a local AI model through an inference server. Popular choices include:

- **Ollama:** Easy to set up, download, and run various open-source models (e.g., Llama 3, Mixtral). It provides a local API endpoint.
- **LM Studio / Jan AI:** Desktop applications that simplify running local models and expose API endpoints.
- **Custom servers:** For highly specific needs, you might set up your own server using libraries like `transformers` or `llama.cpp`.

1. **Configuring Pi as a Custom Provider:** Pi's flexibility allows you to define custom AI model providers. This usually involves modifying Pi's configuration file (e.g., `~/.pi/config.ts` or a project-level `.pi/config.ts`).

   Example (Conceptual `config.ts` for Ollama):

   ```typescript
   // ~/.pi/config.ts (or similar)
   import { Configuration } from '@mariozechner/pi-coding-agent/dist/types';

   const config: Configuration = {
     providers: {
       openai: {
         apiKey: process.env.OPENAI_API_KEY,
         // ... other OpenAI settings
       },
       ollama: {
         // Define a new provider named 'ollama'
         type: 'custom', // Indicate it's a custom provider
         baseUrl: 'http://localhost:11434/v1', // Ollama's default API endpoint
         models: [
           { id: 'llama3', name: 'Llama 3 (Local)', // Add any specific parameters for this model if needed },
           { id: 'codellama', name: 'Code Llama (Local)', }
         ],
         // You might need to specify headers or other auth if your local server requires it
       },
       // ... other Pi configurations
     };

   export default config;
   ```

   > ⚡ **Note:** The exact structure for defining custom providers in Pi's `config.ts` may vary. Always consult the official Pi documentation (`pi.dev/docs/latest/settings`) for the most up-to-date and accurate configuration syntax.

2. **Using the Local Model:** Once configured, you can then tell Pi to use your local model:

   `bash pi --provider ollama --model llama3`

   Or, if you make it your default in `config.ts`, simply `pi`.

Benefits of Local AI with Pi:

- **Privacy:** Your code and data never leave your machine, crucial for proprietary projects.
- **Cost-Effectiveness:** No per-token API charges, making it ideal for high-volume or long-running tasks.
- **Offline Capability:** Work on your code with AI assistance even without an internet connection.
- **Customization:** Fine-tune models locally for very specific domain knowledge or coding styles.

> ⚡ **Real-world insight:** Many development teams are adopting a hybrid approach, using powerful cloud models for complex reasoning and local models for routine code generation, internal documentation, or sensitive code analysis.

By integrating Pi with frameworks like OpenClaw and leveraging local AI models, you create a robust, secure, and highly efficient agentic development environment tailored to your specific needs and constraints.

Debugging, Best Practices, and Real-World Scenarios

Even with a powerful tool like Pi, effective usage requires understanding how to debug issues, adhere to best practices, and apply it to real-world development challenges. This section will equip you with the knowledge to maximize your productivity with Pi.

Debugging Your Agent Interactions

When Pi doesn't behave as expected, debugging involves understanding what the agent is "thinking" and how it's interpreting your prompts.

1. **Verbose Output:** Most CLI tools, including Pi, offer a verbose mode. `bash pi --verbose # or pi -v` This command will often print more detailed

logs, including the full prompts sent to the AI model, the raw responses, and any internal processing steps. This is invaluable for understanding why an AI might be generating unexpected output.

2. **Inspect Agent Logs:** Pi might generate internal logs in a specific directory (e.g., `~/.pi/logs`). Checking these files can reveal errors in extension loading, API communication issues, or internal agent failures.
3. **Check API Provider Logs:** If you're using a cloud provider like OpenAI, check their dashboard for API usage logs. This can help identify issues like:

- **Rate Limits:** You're sending too many requests too quickly.
- **Authentication Errors:** Your API key is invalid or expired.
- **Billing Issues:** Your account might have run out of credits.

1. **Simplify and Isolate:** If a complex prompt is failing, try breaking it down into smaller, simpler requests. This helps isolate where the agent's understanding breaks down. Test your custom extensions in isolation if possible.

> ⚠️ **What can go wrong:** A common debugging challenge is "garbage in, garbage out." If your initial prompt is ambiguous, incomplete, or contains errors, the agent's response will likely be poor. Always strive for clear, specific, and well-structured prompts.

Best Practices for Pi Workflows

To get the most out of Pi, adopt these best practices:

1. **Start Small and Iterate:** Don't ask Pi to rewrite your entire codebase in one go. Start with small, well-defined tasks (e.g., "refactor this single function," "generate tests for this class"). Review its output, then iterate.
2. **Clear and Specific Prompts:** Be explicit. Instead of "Fix this code," try "Refactor `src/utils/data.ts`'s `processData` function to be more performant by using `Map` instead of plain objects for lookups, ensuring type safety with TypeScript, and adding JSDoc comments."
3. **Provide Context:** Pi works best when it has relevant context. Ensure you're running `pi` in the correct project directory, and consider feeding it relevant file contents or diffs when asking for modifications.
4. **Version Control Your Extensions and Templates:** Treat your custom Pi extensions, skills, and prompt templates like any other code. Store them in

version control (Git) so you can track changes, collaborate, and easily revert if needed.

5. **Review All Agent Output:** Never blindly accept code or commands generated by Pi. Always review, understand, and test its suggestions before committing them to your codebase. Pi is an assistant, not an infallible oracle.
6. **Understand Model Capabilities:** Be aware of the strengths and weaknesses of the AI models you're using. A smaller model might be fast but prone to errors, while a larger model might be more accurate but slower and more expensive. Choose wisely for the task.
7. **Manage Costs:** If using cloud models, keep an eye on your API usage. Use `pi --model` to switch to cheaper models for less critical tasks.

Real-World Development Scenarios

Pi can significantly boost productivity across various daily software engineering tasks:

1. **Refactoring Legacy Code:** Ask Pi to identify code smells, suggest refactoring patterns, and even apply them to specific functions or modules.
    - Prompt example: "Analyze `src/legacy/old_service.js` for potential performance bottlenecks and suggest modern JavaScript refactorings for the `calculateTotal` function."
2. **Generating Boilerplate and Tests:** Quickly spin up new components, API endpoints, or comprehensive unit tests.
    - Prompt example: "Create a new Express.js route in `src/routes/users.ts` for `GET /users/:id` that fetches a user from a mock database. Also, generate Jest unit tests for this route."
3. **Debugging and Error Analysis:** Paste error messages, stack traces, and relevant code snippets, and ask Pi to diagnose the problem and suggest fixes.
    - Prompt example: "I'm getting this error: `TypeError: Cannot read properties of undefined (reading 'name')` in `src/components/UserProfile.tsx` at line 45. Here's the code for `UserProfile.tsx`. What's the likely cause and how can I fix it?"

4. **Learning New APIs/Frameworks:** Ask Pi to explain concepts, provide examples, or even write small proof-of-concept snippets for unfamiliar technologies.
    - Prompt example: "Explain how to use React's `useContext` hook and provide a simple example of passing a theme object down a component tree."
5. **Code Review Pipelines:** Integrate Pi into your CI/CD pipeline or use it interactively to perform automated code reviews, checking for style, security, or logical errors.
    - Workflow: Generate a `git diff` for a pull request, feed it to Pi with a "Code Reviewer" prompt template, and summarize its findings.

By integrating Pi strategically into these scenarios, you can offload cognitive burden, accelerate development, and maintain high code quality.

🧠 Check Your Understanding

- What is the primary advantage of using Pi for multi-agent workflows compared to a single, more powerful agent?
- Explain two key benefits of leveraging local AI models with Pi over purely cloud-based solutions.
- If you're encountering `EACCES` errors during `npm install -g @mariozechner/pi-coding-agent`, what is the most common solution?

⚡ Mini Task

- Create a new directory, initialize a Node.js project, and install Pi Coding Agent. Then, start a Pi session and ask it to explain the difference between `let` and `const` in JavaScript.

🚀 Scenario

- You are a lead developer on a new project that requires strict data privacy and compliance. Your team wants to use Pi Coding Agent for generating boilerplate code and unit tests. You have access to powerful open-source large language models (LLMs) that can run locally. Outline the steps you would take to set up Pi to use these local LLMs, ensuring that no code leaves

your development environment. What are the key configuration points you'd need to address?

What to Build Next

Congratulations on mastering Pi Coding Agent! You've learned how to install it, configure models, manage workflows, customize its behavior, and leverage advanced agentic patterns. Here are three concrete ideas to continue your journey and build upon your new skills:

1. **Automated Documentation Generator:** Build a custom Pi extension that, when run in a project directory, scans your codebase (e.g., TypeScript files), identifies functions and classes, and generates initial JSDoc or TSDoc comments. You could integrate this with a prompt template that guides Pi on the desired documentation style.
2. **Personalized Code Style Enforcer:** Create a multi-agent workflow where one agent identifies deviations from your team's specific coding style guide (e.g., using an `AGENTS.md` entry), and another agent suggests and applies the necessary fixes. This could be triggered by a pre-commit hook that uses Pi.
3. **CLI Project Scaffolder:** Develop a sophisticated Pi extension that can scaffold entire project structures based on user input. For example, `pi scaffold react-app-with-auth` could prompt for authentication method, database, and then generate all necessary files, install dependencies, and even write basic CRUD operations using your preferred framework.

📌 TL;DR

- Pi Coding Agent is a minimal, extensible CLI AI assistant for software engineering tasks.
- It streamlines workflows through agentic capabilities, living directly in your terminal.
- Installation is via `npm`, requiring an AI provider API key (e.g., OpenAI) or local model setup.
- Sessions, branching, and dynamic model switching enable flexible and iterative problem-solving.
- Customization via TypeScript extensions, skills, and prompt templates allows for tailored automation.

- Advanced uses include self-extending agents and multi-agent workflows (e.g., `AGENTS.md` for specialists).
- Pi integrates with broader frameworks like OpenClaw and supports local AI models for privacy and cost control.

🧠 Core Flow

1. Install Pi Coding Agent globally via npm.
2. Configure AI model provider API keys as environment variables.
3. Start a Pi session in your project directory.
4. Interact with Pi, leveraging sessions, branching, and model switching.
5. Extend Pi's capabilities with custom TypeScript extensions, skills, and prompt templates.
6. Build advanced workflows using self-extension and multi-agent configurations.
7. Integrate with external tools like OpenClaw and local AI inference servers.

🚀 Key Takeaway

Pi Coding Agent empowers developers to transform their terminal into an intelligent, extensible, and collaborative AI-driven environment, enabling unprecedented efficiency and automation in daily software engineering tasks by bringing agentic capabilities directly to the code.