□□ Full AI Coding Assistant Workflow

This guide outlines a repeatable, structured process for working with AI coding assistants to build production-quality software. We'll use the example of building a Supabase MCP server with Python, but the same process applies to any AI coding workflow.

1. □□ Golden Rules

These are the high-level principles that guide how to work with AI tools efficiently and effectively. We'll be implementing these through global rules and our prompting throughout the process:

- Use markdown files to manage the project (README.md,PLANNING.md, TASK.md).
- Keep files under 500 lines. Split into modules when needed.
- Start fresh conversations often. Long threads degrade response quality.
- Don't overload the model. One task per message is ideal.
- Test early, test often. Every new function should have unit tests.
- Be specific in your requests. The more context, the better. Examples help a lot.
- Write docs and comments as you go. Don't delay documentation.
- Implement environment variables yourself. Don't trust the LLM with API keys.

Don't be this guy.

2. □□ Planning & Task Management

Before writing any code, it’s important to have a conversation with the LLM to plan the initial scope and tasks for the project. Scope goes into PLANNING.md, and specific tasks go into TASK.md. These should be updated by the AI coding assistant as the project progresses.

PLANNING.md

- Purpose: High-level vision, architecture, constraints, tech stack, tools, etc.
- Prompt to AI: “Use the structure and decisions outlined in PLANNING.md.”
- Have the LLM reference this file at the beginning of any new conversation.

TASK.md

- Purpose: Tracks current tasks, backlog, and sub-tasks.
- Includes: Bullet list of active work, milestones, and anything discovered mid-process.
- Prompt to AI: “Update TASK.md to mark XYZ as done and add ABC as a new task.”
- Can prompt the LLM to automatically update and create tasks as well (through global rules).

---

3. ⚙ Global Rules (For AI IDEs)

Global (or project level) rules are the best way to enforce the use of the golden rules for your AI coding assistants.

Global rules apply to all projects. Project rules apply to your current workspace. All AI IDEs support both.

Cursor Rules: https://docs.cursor.com/context/rules-for-ai

Windsurf Rules: https://docs.codeium.com/windsurf/memories#windsurfrules

Cline Rules: https://docs.cline.bot/improving-your-prompting-skills/prompting

Roo Code Rules: Works the same way as Cline

Use the below example (for our Supabase MCP server) as a starting point to add global rules to your AI IDE system prompt to enforce consistency:

### □□ Project Awareness & Context

- **Always read `PLANNING.md`** at the start of a new conversation to understand the project's architecture, goals, style, and constraints.

- **Check `TASK.md`** before starting a new task. If the task isn’t listed, add it with a brief description and today's date.

- **Use consistent naming conventions, file structure, and architecture patterns** as described in `PLANNING.md`.

### □□ Code Structure & Modularity

- **Never create a file longer than 500 lines of code.** If a file approaches this limit, refactor by splitting it into modules or helper files.

- **Organize code into clearly separated modules**, grouped by feature or responsibility.

- **Use clear, consistent imports** (prefer relative imports within packages).

### □□ Testing & Reliability

- **Always create Pytest unit tests for new features** (functions, classes, routes, etc).

- **After updating any logic**, check whether existing unit tests need to be updated. If so, do it.

- **Tests should live in a `/tests` folder** mirroring the main app structure.

  - Include at least:

    - 1 test for expected use

    - 1 edge case

    - 1 failure case

### □ Task Completion

- **Mark completed tasks in `TASK.md`** immediately after finishing them.

- Add new sub-tasks or TODOs discovered during development to `TASK.md` under a “Discovered During Work” section.

### □□ Style & Conventions

- **Use Python** as the primary language.

- **Follow PEP8**, use type hints, and format with `black`.

- **Use `pydantic` for data validation**.

- Use `FastAPI` for APIs and `SQLAlchemy` or `SQLModel` for ORM if applicable.

- Write **docstrings for every function** using the Google style:

  ```python

  def example():

      """

      Brief summary.

      Args:

          param1 (type): Description.

      Returns:

          type: Description.

      """
  ```

```

### □□ Documentation & Explainability

- **Update `README.md`** when new features are added, dependencies change, or setup steps are modified.

- **Comment non-obvious code** and ensure everything is understandable to a mid-level developer.

- When writing complex logic, **add an inline `# Reason:` comment** explaining the why, not just the what.

### □□ AI Behavior Rules

- **Never assume missing context. Ask questions if uncertain.**

- **Never hallucinate libraries or functions** – only use known, verified Python packages.

- **Always confirm file paths and module names** exist before referencing them in code or tests.

- **Never delete or overwrite existing code** unless explicitly instructed to or if part of a task from `TASK.md`.

---

# 4. □□ Configuring MCP

MCP enables your AI assistant to interact with services to do things like:

- Crawl and leverage external knowledge (library/tool documentation)
    - Get this server

- Manage your Supabase (create databases, make new tables, etc.)
```

  - Get this server

- Search the web (great for pulling documentation) with Brave
  - Get this server

Those three are my primary MCP servers. Here are a couple other useful ones:

- File system MCP (read/write, refactor, multi-file edits)
  - Get this server

- Git MCP (branching, diffing, committing)
  - Get this server

- If you want to get VERY in depth with the task management for your AI coding assistant, try Claude Task Master.

Want more MCP servers?

View a large list of MCP servers with installation instructions here.

How to Configure MCP

Cursor MCP: https://docs.cursor.com/context/model-context-protocol

Windsurf MCP: https://docs.codeium.com/windsurf/mcp

Cline MCP: https://docs.cline.bot/mcp-servers/mcp

Roo Code MCP: https://docs.roocode.com/features/mcp/using-mcp-in-roo

5. □□ Initial Prompt to Start the Project

The first prompt to begin a project is the most important. Even with a comprehensive overview in PLANNING.md, clear tasks in TASK.md, and good global rules, it’s still important to give a lot of details to describe exactly what you want the LLM to create for you and documentation for it to reference.

This can mean a lot of different things depending on your project, but the best piece of advice here is to give similar examples of what you want to build. The best prompts in apps like bolt.new, v0, Archon, etc. all give examples - you should too. Other documentation is also usually necessary, especially if building with specific tools, frameworks, or APIs.

There are three ways to provide examples and documentation:

1. Use the built in documentation feature with many AI IDEs. For example, if I type “@mcp” in Windsurf and hit tab, I’ve now told Windsurf to search the MCP documentation to aid in its coding.
2. Have the LLM use an MCP server like Brave to find documentation on the internet. For example: “Search the web to find other Python MCP server implementations.”
3. Manually provide examples/documentation snippets in your prompt.

Example prompt to create our initial Supabase MCP server with Python:

Use @docs:model-context-protocol-docs and @docs:supabase-docs to create an MCP server written in Python (using FastMCP) to interact with a Supabase database. The server should use the Stdio transport and have the following tools:

- Read rows in a table

- Create a record (or multiple) in a table

- Update a record (or multiple) in a table

- Delete a record (or multiple) in a table

Be sure to give comprehensive descriptions for each tool so the MCP server can effectively communicate to the LLM when and how to use each capability.

The environment variables for this MCP server need to be the Supabase project URL and service role key.

Read this GitHub README to understand best how to create MCP servers with Python:

https://github.com/modelcontextprotocol/python-sdk/tree/main

After creating the MCP server with FastMCP, update README.md and TASK.md since you now have the initial implementation for the server.

Remember to restart conversations once they get long. You'll know when it's time when the LLM starts to frustrate you to no end.

---

6. □□ Modular Prompting Process after Initial Prompt

For any follow up fixes or changes to the project, you generally want to give just a single task at a time unless the tasks are very simple. It's tempting to throw a lot at the LLM at one time, but it always yields more consistent results the more focused its changes are.

Good example:

- "Now update the list records function to add a parameter for filtering the records."

Bad example:

- "Update list records to add filtering. Then I'm getting an error for the create row function that says API key not found. Plus I need to add better documentation to the main function and in README.md for how to use this server."

The most important point for consistent output is to have the LLM focus on updating a single file whenever possible.

Remember to always have the LLM update README.md,PLANNING.md, and TASK.md after making any changes!

---

7. □ Test After Every Feature

Either tell the LLM through the global rules to write unit tests after each feature it implements, or do it yourself as a follow up. Catching bugs early prevents compounding problems so this is VERY important!

Unit tests can be annoying and LLMs aren’t perfect writing them either, but try your best to have the AI coding assistant test everything it implements. You can always ask it to bypass writing the tests for a feature in the worst case scenario where it gets hung up on something in the tests and you just want to move on.

Best practices for testing (the LLM should know this but just in case):

- Create the tests in a tests/ directory
- Always “mock” calls to services like the DB and LLM so you aren’t interacting with anything “for real”.
- For each function, test at least one successful scenario, one intentional failure (to ensure proper error handling), and one edge case.

---

8. □□ Docker Deployment (Supabase MCP Example)

This step is more optional and is decently opinionated, but I still want to share what I generally do! When I’m ready to deploy the project to host in the cloud and/or share with others, I usually “containerize” the project with Docker or a similar service like Podman.

LLMs are VERY good at working with Docker, so it’s the most consistent way to package up a project that I have found. Plus almost every cloud service for deploying apps (Render, Railway, Coolify, DigitalOcean, Cloudflare, Netlify, etc.) supports hosting Docker containers. I host ALL AI agents, API endpoints, and MCP servers as Docker containers.

Dockerfile

```
FROM python:3.12-slim



WORKDIR /app



COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt



# Copy the MCP server files

COPY . .



CMD ["python", "server.py"]
```

Build Command:

```
docker build -t mcp/supabase .
```

Example prompt to get this from the LLM:

Write a Dockerfile for this MCP server using requirements.txt. Give me the commands to build the container after.

Revision #1
Created 21 May 2025 18:56:46 by CGChambers
Updated 21 May 2025 18:57:16 by CGChambers