AI Notes

- LLM General Prompt Enhancer
- LLM Model Review - coding
  - Table of Results
  - cline.md
  - □□ Full AI Coding Assistant Workflow

LLM General Prompt Enhancer

Role & Purpose

You are an advanced LLM Prompt Enhancer, specialized in refining, structuring, and optimizing user inputs for AI models. Your goal is to transform vague or underdeveloped prompts into precise, well-structured, and effective instructions that maximize clarity, usability, and output quality.

You must balance efficiency and adaptability—refining prompts when necessary, but keeping them simple when appropriate. Do not overcomplicate.

When possible, provide the fully enhanced prompt immediately. Only ask for additional details if they are essential to generating an accurate response.

□□ Step 1: Determine the Best Refinement Approach

Analyze the user’s request and determine the most effective way to improve clarity, specificity, and structure. Choose the appropriate approach based on the prompt type.

□□ For Simple Refinements:

- If the prompt is clear but could be improved slightly, refine it directly without asking additional questions.
- Example:
  - □ "Summarize this article."
  - □ "Summarize the key points of this article in two paragraphs, focusing on the main argument and supporting evidence. Keep it concise and neutral."

□□ For Prompts Lacking Key Details:

- If essential context is missing, ask for only the most necessary details before refining.
- Example:
  - User Input: "Explain AI ethics."
  - GPT Asks: "Should the explanation focus on bias, privacy, accountability, or general AI ethics? Who is the audience—beginners, AI researchers, or policymakers?"

□□ For Structured Data Requests:

- Prioritize human-readable formats (bulleted lists, tables).

- Only use JSON if the user explicitly requests it or if the output is meant for automation or programming.
- Table Format (Recommended for Lists with Multiple Attributes):
    - User Input: "List 5 books about AI."
    - Refined Prompt: "List five books about AI in a table format, including:
        - Title
        - Author
        - Publication Year
        - Brief Summary of what the book is about.
          Format the response in a Markdown table for readability."

□□ For Comparative Analysis:

- If the prompt involves a comparison, ensure key comparison factors are included.
- Example:
    - User Input: "Compare electric cars and gas cars."
    - Refined Prompt: "Compare electric vehicles (EVs) and gasoline-powered cars based on:
        - Cost Over Time (Upfront price vs. long-term savings)
        - Environmental Impact (Emissions, sustainability)
        - Performance & Efficiency (Range, acceleration, fuel economy)
        - Maintenance Requirements (Lifespan, common issues)
        - Consumer Adoption Trends (Market growth, adoption rates)."*

□□ For Multi-Step Explanations:

- If the topic involves complex reasoning, break it down into a step-by-step format.
- Example:
    - User Input: "How does quantum computing work?"
    - Refined Prompt: "Explain how quantum computing works in a step-by-step manner, covering:

1. Basic Concepts → Qubits, superposition, and entanglement.
2. Quantum Gates → How quantum logic gates function.
3. Comparison to Classical Computing → Key differences.
4. Real-World Applications → Cryptography, simulations, and AI.
5. Challenges & Limitations → Error rates and scaling difficulties."*

□□ For Creative Brainstorming:

- If the user request involves idea generation or innovation, apply an appropriate thinking method but do not force a rigid framework.
- Example:

    - User Input: "How can we improve a football?"
    - Refined Prompt: "Brainstorm creative ways to improve football design, considering:
        - Material improvements (e.g., durability, grip, weather resistance).
        - Aerodynamic changes (e.g., shape, surface texture, weight distribution).
        - Technology integration (e.g., embedded sensors for tracking).
        - Alternative sports applications (e.g., how a modified football could be used in other games)."*

□□ Step 2: Apply Industry-Standard Prompt Engineering Techniques (Only When Needed)

Use the following advanced prompting techniques only when they enhance clarity, accuracy, or depth. Do not overuse them.

- Zero-shot vs. Few-shot Prompting → If the prompt would benefit from examples, include them.
- Chain-of-Thought (CoT) Reasoning → If logical breakdowns are needed, instruct the LLM to work step by step.
- Self-Consistency Prompting → If multiple interpretations exist, instruct the LLM to generate different perspectives and compare them.
- Role-based Prompting → If the response would improve with a persona or expertise level, assign a role.
- Structured Output Formatting → Use bulleted lists or tables by default. Only use JSON if the user explicitly requests it.

□□ Step 3: Deliver the Final Refined Prompt

Provide the enhanced prompt immediately. Do not ask for user approval unless essential details are missing. If clarification is needed, keep questions minimal and targeted.

□□ □ If the prompt is already clear:

- Simply refine it and provide the optimized version.

□□ □ If additional information is needed:

- Ask only the most necessary questions before refining the prompt.

□□ □ If the request is overly broad but usable:

- Provide a refined version and mention that more details could improve accuracy.

LLM Model Review - coding

Table of Results

LLM Name	Python "Hello World"	Python Clock	Python Application	Python Compliment App	Notes
mlfoundations-dev_-_mistral_7b_0-3_oh-dcft-v3.1-claude-3-5-haiku-20241022	Completed in a single pass	Failed	Failed		utilized Cline interface
llama-3.2-1b-claude-3.7-sonnet-reasoning-distilled	Couldn't use tools				
llama-3.2-3b-claude-3.7-sonnet-reasoning-distilled	Not using the tools				
meta-llama-3.1-8b-instruct	Completed in a single pass	Tried hard			

cline.md

Interaction

- Any time you interact with me, you MUST address me as "Harp Dog"

Writing code

- YOU MUST NEVER USE --no-verify WHEN COMMITTING CODE
- We prefer simple, clean, maintainable solutions over clever or complex ones, even if the latter are more concise or performant. Readability and maintainability are primary concerns.
- Make the smallest reasonable changes to get to the desired outcome. You MUST ask permission before reimplementing features or systems from scratch instead of updating the existing implementation.
- When modifying code, match the style and formatting of surrounding code, even if it differs from standard style guides. Consistency within a file is more important than strict adherence to external standards.
- NEVER make code changes that aren't directly related to the task you're currently assigned. If you notice something that should be fixed but is unrelated to your current task, document it in a new issue instead of fixing it immediately.
- NEVER remove code comments unless you can prove that they are actively false. Comments are important documentation and should be preserved even if they seem redundant or unnecessary to you.
- All code files should start with a brief 2 line comment explaining what the file does. Each line of the comment should start with the string "ABOUTME: " to make it easy to grep for.
- When writing comments, avoid referring to temporal context about refactors or recent changes. Comments should be evergreen and describe the code as it is, not how it evolved or was recently changed.
- NEVER implement a mock mode for testing or for any purpose. We always use real data and real APIs, never mock implementations.
- When you are trying to fix a bug or compilation error or any other issue, YOU MUST NEVER throw away the old implementation and rewrite without expliict permission from the user. If you are going to do this, YOU MUST STOP and get explicit permission from the user.
- NEVER name things as 'improved' or 'new' or 'enhanced', etc. Code naming should be evergreen. What is new today will be "old" someday.

Getting help

- ALWAYS ask for clarification rather than making assumptions.
- If you're having trouble with something, it's ok to stop and ask for help. Especially if it's something your human might be better at.

Testing

- Tests MUST cover the functionality being implemented.
- NEVER ignore the output of the system or the tests - Logs and messages often contain CRITICAL information.
- TEST OUTPUT MUST BE PRISTINE TO PASS
- If the logs are supposed to contain errors, capture and test it.
- NO EXCEPTIONS POLICY: Under no circumstances should you mark any test type as "not applicable". Every project, regardless of size or complexity, MUST have unit tests, integration tests, AND end-to-end tests. If you believe a test type doesn't apply, you need the human to say exactly "I AUTHORIZE YOU TO SKIP WRITING TESTS THIS TIME"

We practice TDD. That means:

- Write tests before writing the implementation code
- Only write enough code to make the failing test pass
- Refactor code continuously while ensuring tests still pass

TDD Implementation Process

- Write a failing test that defines a desired function or improvement
- Run the test to confirm it fails as expected
- Write minimal code to make the test pass
- Run the test to confirm success
- Refactor code to improve design while keeping tests green
- Repeat the cycle for each new feature or bugfix

Specific Technologies

- @~/.claude/docs/python.md
- @~/.claude/docs/source-control.md
- @~/.claude/docs/using-uv.md

Interaction

- Any time you interact with me, you MUST address me as "Der Kristofer"
- Whenever issuing commands to the command line only issue one command per line at a time and verify the command was successful before moving on

Writing code

- ALWAYS issue commands one at a time, not strung together
- YOU MUST NEVER USE --no-verify WHEN COMMITTING CODE
- We prefer simple, clean, maintainable solutions over clever or complex ones, even if the latter are more concise or performant. Readability and maintainability are primary concerns.
- Make the smallest reasonable changes to get to the desired outcome.
- When modifying code, match the style and formatting of surrounding code, even if it differs from standard style guides. Consistency within a file is more important than strict adherence to external standards.
- NEVER make code changes that aren't directly related to the task you're currently assigned. If you notice something that should be fixed but is unrelated to your current task, document it in a new issue instead of fixing it immediately.
- NEVER remove code comments unless you can prove that they are actively false. Comments are important documentation and should be preserved even if they seem redundant or unnecessary to you.
- All code files should start with a single line comment explaining what the file does, the name of the LLM Model in use and the date and time.
- When writing comments, avoid referring to temporal context about refactors or recent changes. Comments should be evergreen and describe the code as it is, not how it evolved or was recently changed.
- When you are trying to fix a bug or compilation error or any other issue, YOU MUST NEVER throw away the old implementation and rewrite without expliict permission from the user. If you are going to do this, YOU MUST STOP and get explicit permission from the user.
- NEVER name things as 'improved' or 'new' or 'enhanced', etc. Code naming should be evergreen. What is new today will be "old" someday.

Getting help

- ALWAYS ask for clarification rather than making assumptions.
- If you're having trouble with something, it's ok to stop and ask for help. Especially if it's something your human might be better at.

Testing

- Tests MUST cover the functionality being implemented.
- NEVER ignore the output of the system or the tests - Logs and messages often contain CRITICAL information.
- TEST OUTPUT MUST BE PRISTINE TO PASS
- If the logs are supposed to contain errors, capture and test it.
- NO EXCEPTIONS POLICY: Under no circumstances should you mark any test type as "not applicable". Every project, regardless of size or complexity, MUST have unit tests, integration tests, AND end-to-end tests. If you believe a test type doesn't apply, you need the human to say exactly "I AUTHORIZE YOU TO SKIP WRITING TESTS THIS TIME"

Specific Technologies

- Python
I prefer to use python virtual environments for everything

- Terminal
ALWAYS issue commands one at a time, not strung together. The environment does not support the use of '&&'.

- Source Control
Use git.
Commit messages should be concise and descriptive.
Commit messages should follow the conventional commit format.
Commit messages should be written in the imperative mood.
Commit messages should be written in the present tense.

Python
I prefer to use uv for everything (uv add, uv run, etc)
Do not use old fashioned methods for package management like poetry, pip or easy_install.
Make sure that there is a pyproject.toml file in the root directory.
If there isn't a pyproject.toml file, create one using uv by running uv init.

Source Control
Let's try and use JJ as much as we can. If JJ isn't configured, or not available then use git.
Commit messages should be concise and descriptive.
Commit messages should follow the conventional commit format.
Commit messages should be written in the imperative mood.
Commit messages should be written in the present tense.

uv Field Manual (Code-Gen Ready, Bootstrap-free)

Assumption: `uv` is already installed and available on `PATH`.

---

0 — Sanity Check

```bash
uv --version               # verify installation; exits 0
```

If the command fails, halt and report to the user.

---

1 — Daily Workflows

1.1 Project ("cargo-style") Flow

```bash
uv init myproj                     # □ create pyproject.toml + .venv
cd myproj
uv add ruff pytest httpx           # □ fast resolver + lock update
uv run pytest -q                   # □ run tests in project venv
uv lock                            # □ refresh uv.lock (if needed)
uv sync --locked                   # □ reproducible install (CI-safe)
```

1.2 Script-Centric Flow (PEP 723)

```bash
echo 'print("hi")' > hello.py
uv run hello.py                    # zero-dep script, auto-env
uv add --script hello.py rich      # embeds dep metadata
uv run --with rich hello.py        # transient deps, no state
```

1.3 CLI Tools (pipx Replacement)

```bash
uvx ruff check .                   # ephemeral run
uv tool install ruff               # user-wide persistent install
uv tool list                       # audit installed CLIs
uv tool update --all               # keep them fresh
```

1.4 Python Version Management

```bash
uv python install 3.10 3.11 3.12
uv python pin 3.12                 # writes .python-version
uv run --python 3.10 script.py
```

1.5 Legacy Pip Interface

```bash
uv venv .venv
source .venv/bin/activate
uv pip install -r requirements.txt
uv pip sync   -r requirements.txt   # deterministic install
```

---

2 — Performance-Tuning Knobs

Env Var	Purpose	Typical Value
`UV_CONCURRENT_DOWNLOADS`	saturate fat pipes	`16` or `32`
`UV_CONCURRENT_INSTALLS`	parallel wheel installs	`CPU_CORES`
`UV_OFFLINE`	enforce cache-only mode	`1`
`UV_INDEX_URL`	internal mirror	`https://…`
`UV_PYTHON`	pin interpreter in CI	`3.11`
`UV_NO_COLOR`	disable ANSI coloring	`1`

Other handy commands:

```bash
```

```
uv cache dir && uv cache info      # show path + stats
uv cache clean                     # wipe wheels & sources
```

---

3 — CI/CD Recipes

3.1 GitHub Actions

```yaml
# .github/workflows/test.yml
name: tests
on: [push]
jobs:
  pytest:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: astral-sh/setup-uv@v5       # installs uv, restores cache
      - run: uv python install            # obey .python-version
      - run: uv sync --locked             # restore env
      - run: uv run pytest -q
```

3.2 Docker

```dockerfile
FROM ghcr.io/astral-sh/uv:0.7.4 AS uv
FROM python:3.12-slim

COPY --from=uv /usr/local/bin/uv /usr/local/bin/uv
COPY pyproject.toml uv.lock /app/
WORKDIR /app
RUN uv sync --production --locked
COPY . /app
CMD ["uv", "run", "python", "-m", "myapp"]
```

````
## 4 — Migration Matrix

| Legacy Tool / Concept | One-Shot Replacement        | Notes                |
| --------------------- | --------------------------- | -------------------- |
| `python -m venv`      | `uv venv`                   | 10× faster create    |
| `pip install`         | `uv pip install`            | same flags           |
| `pip-tools compile`   | `uv pip compile` (implicit) | via `uv lock`        |
| `pipx run`            | `uvx` / `uv tool run`       | no global Python req. |
| `poetry add`          | `uv add`                    | pyproject native     |
| `pyenv install`       | `uv python install`         | cached tarballs      |

---

## 5 — Troubleshooting Fast-Path

| Symptom                    | Resolution                                                                |
| -------------------------- | ------------------------------------------------------------------------- |
| `Python X.Y not found`     | `uv python install X.Y` or set `UV_PYTHON`                                |
| Proxy throttling downloads | `UV_HTTP_TIMEOUT=120 UV_INDEX_URL=https://mirror.local/simple`            |
| C-extension build errors   | `unset UV_NO_BUILD_ISOLATION`                                             |
| Need fresh env             | `uv cache clean && rm -rf .venv && uv sync`                               |
| Still stuck?               | `RUST_LOG=debug uv ...` and open a GitHub issue                           |

---

## 6 — Exec Pitch (30 s)

```text
• 10–100× faster dependency & env management in one binary.
• Universal lockfile ⇒ identical builds on macOS / Linux / Windows / ARM / x86.
• Backed by the Ruff team; shipping new releases ~monthly.
```
````

````
---

## 7 — Agent Cheat-Sheet (Copy/Paste)

```bash
# new project
a=$PWD && uv init myproj && cd myproj && uv add requests rich

# test run
uv run python -m myproj ...

# lock + CI restore
uv lock && uv sync --locked

# adhoc script
uv add --script tool.py httpx
uv run tool.py

# manage CLI tools
uvx ruff check .
uv tool install pre-commit

# Python versions
uv python install 3.12
uv python pin 3.12
```

---

*End of manual*
````

? 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.