

Prompt Fu Failure: How a Broken Script Revealed a Better Architecture

Context for the Curious Book Reader

What begins as a routine journal entry about a planned coding task—a Saturday afternoon refactor—quickly spirals into something more. The developer, using a custom script to package up his codebase for an AI assistant, hits an unexpected and show-stopping error. The very tool he built to streamline his workflow has broken. This entry captures the real-time process of debugging not just the code, but the method of collaboration with the AI itself. It's a raw look at the frustrating, yet ultimately illuminating, detours that happen in modern software development, revealing how a simple bug fix can lead to a profound architectural insight and a stronger path forward.

Technical Journal Entry Begins

Alright now the trick is extract and externalize Pipulate without breaking everything and without kicking off a major refactoring. This is a job for the Prompt Fu process so first I'm going to explain some of the criteria and then I'm going to figure out which files to put in context for the AI to help me examine and come up with a plan. There's a few things that I notice.

The first is that the Pipulate class is a huge chunk of server.py and so the externalization is right off the bat going to have the advantage of bringing server.py down to a better size, which immediately reminds me that I did recent work when I created the imports/ folder and extracted the server logging functions into imports/server_logging.py and imports/stream_orchestrator.py. Wait a minute! I also have imports/mcp_orchestrator.py. I do believe stream_orchestrator supersedes mcp_orchestrator but I'm not sure.

Can you take a look and help? This will be my first test of Prompt Fu system that puts everything in the one large XML context before I tackle the larger job of externalizing Pipulate we start with this smaller job of analyzing whether I have some dead code I need to retire stream_orchestrator.py. Wait a minute! I also have imports/mcp_orchestrator.py. I do believe stream_orchestrator supersedes mcp_orchestrator but I'm not sure. Can you take a look and help? This will be my first test of Prompt Fu system that puts everything in the one large XML context before I tackle the larger job of externalizing Pipulate we start with this smaller job of analyzing whether I have some dead code I need to retire.

Uh oh! A nested project. Something is going wrong with my Prompt Fu process and Gemini keeps coming back saying "Something went wrong" when I try to submit the enormous XML payloads. It starts like it's accepting them but it pushes back. Now something has changed and it might be from the Gemini side but just as likely from my side as I have been doing some changes in the system recently.

How do I show the last time a particular file changed in a git history?

Gemini: To see the last time a particular file was changed in your Git history, you can use the git log command. This command provides a detailed history of commits for a specified file.

Finding the Last Change

To view the most recent commit that affected a specific file, execute the following command in your terminal, replacing [file-path] with the actual path to your file:

```
git log -1 [file-path]
```

This command will display the full commit details, including the commit hash, author, date, and the commit message for the last change made to that file.

Customizing the Output

For a more concise and readable output, you can format the log. To display the commit hash, the relative date of the change, and the commit message, use:

```
git log -1 --pretty=format:"%h - %ar : %s" [file-path]
```

Here's a breakdown of the format specifiers:

- %h: Abbreviated commit hash
- %ar: Author date, relative (e.g., "2 days ago")
- %s: Subject (the commit message)

Viewing the Actual Changes

If you want to see the specific changes made to the file in that last commit, you can add the -p or --patch flag:

```
git log -1 -p [file-path]
```

This will show the diff of the file, highlighting the lines that were added or removed in the last modification.

MikeLev.in: Okay, once I have the git commit how do I check out that state of only that file?

Gemini: You can checkout the state of a single file from a specific commit using the git checkout command.

Checking Out a Single File

To restore a file to its state from a particular commit, use the following command structure. You'll need the commit hash (the unique identifier for the commit) and the path to the file.

```
git checkout [commit-hash] -- [file-path]
```

For example, if you want to revert the file src/app.js to its state in the commit with the hash a1b2c3d, you would run:

```
git checkout a1b2c3d -- src/app.js
```

Key things to remember:

- --: The double dash is important. It separates the commit hash from the file path. This prevents any ambiguity if your file path could be mistaken for a branch name.

- **Commit Hash:** You can use the full commit hash or a shortened, unique version of it.
- **Working Directory:** This command will modify the file in your **current working directory**. The change will be unstaged, meaning you'll see it when you run `git status`. You can then decide to commit this reverted version of the file or make further changes.
- **No Branch Change:** This action does **not** switch your entire repository to the old commit. Your HEAD will remain on your current branch; only the specified file is affected. 📁
- **Staging the Change:** If you want to immediately stage the reverted file for the next commit, you can simply `git add [file-path]` after running the checkout command.

MikeLev.in: Is that the same to rolling that file back to the older version? Shouldn't it show up on a `git status` or `git diff`? Does it really change the file on the drive or does it need something like `git reset --hard [hash]` equivalent for just that file?

Gemini: Yes, that is the same as rolling that file back to the older version. It does change the file on your drive, and you're right, it should then show up when you run `git status` or `git diff`.

What git checkout Does to a Single File

When you run `git checkout [commit-hash] -- [file-path]`, Git performs a very specific action:

1. **Finds the old version:** It goes back to the specified commit in its history.
2. **Replaces the current file:** It takes the version of the file as it existed in that old commit and uses it to overwrite the file in your current working directory. 📝✏️

So, **yes, it absolutely changes the actual file on your drive**. This action is the direct equivalent of "rolling back" or "reverting" a single file to a previous state without affecting the rest of your project.

How it Appears in git status and git diff

Because the file in your working directory has been modified, Git will immediately recognize the change.

- **git status:** When you run `git status`, Git compares your working directory to the staging area (the index). Since you just overwrote the file with an older version, Git will see that the file is different from what's staged (and likely different from your last commit) and list it under **"Changes not staged for commit"**:
On branch main
Changes not staged for commit: (use "`git add <file>...`" to update what will be committed) (use "`git restore <file>...`" to discard changes in working directory)
modified: your-file.js
- **git diff:** If you run `git diff`, Git will show you the difference between the version of the file in your working directory (the old version you just checked out) and the version in the staging area. This will effectively show you the changes that were made between the old commit and your current state.

The Difference from git reset --hard

Your intuition about `git reset --hard` is on the right track, but there's a crucial difference.

- `git reset --hard [hash]`: This command is a powerful and potentially destructive operation that works on the **entire repository**. It resets your HEAD to a specific commit, overwriting your staging area **and your entire working directory** to match that commit. Any uncommitted changes are lost forever. It's like a time machine for your whole project. 💥
- `git checkout [hash] -- [file-path]`: This is a precise, surgical operation. It only affects the **single file** you specify. It's the safe and correct way to "reset" just one file without touching anything else. It's the "file-level" equivalent you were thinking of, but it uses the checkout command instead of reset.

MikeLev.in: Okay, I had the wrong hash. I have the right one and now git status does show that the file has been modified but git diff shows nothing. Does that mean that the changes are automatically added and that I need to use git show to see the restored changes?

Gemini: No, that doesn't mean the changes are automatically added. It means the version of the file you checked out is identical to the version in your staging area (also called the index).

The Three Trees of Git

To understand this, you need to think about Git's "three trees":

1. **HEAD:** The last commit in your current branch.
2. **Index (Staging Area):** The set of files that will be included in your *next* commit.
3. **Working Directory:** The actual files on your hard drive.

By default, git diff shows the differences between your **Working Directory** and your **Index**. If you haven't used git add on the file since your last commit, your Index still contains the version of the file from the last commit (HEAD).

Why git diff is Empty

When git status shows the file as "modified," it's telling you that the file in your **Working Directory** is different from the version in **HEAD**.

However, when git diff shows nothing, it's telling you there's no difference between your **Working Directory** and the **Index**.

This can happen in one specific scenario: the "right" hash you used to check out the file was actually the hash of the most recent commit (HEAD). So, you effectively "restored" the file to the state it was already in, which is why git diff is empty while git status still notes a modification relative to the last commit.

The Right Command to See the Changes

To see the difference between the file in your working directory and the version from your last commit (HEAD), you need to be more specific.

Run this command:

```
git diff HEAD your-file.js
```

This explicitly tells Git to compare the version in your working directory with the version in HEAD, and it will show you the changes you expect to see.

What about git show?

git show is not the right command here. git show is used to inspect Git objects like commits, tags, and blobs. Running git show by itself will simply display the contents and metadata of the last commit (HEAD). It doesn't show you uncommitted changes in your working directory.

An Unexpected Roadblock: When the Tools Change

MikeLev.in: Hmmm, okay wow I went back to various states of my Prompt Fu system and generated the XML payload and have confirmed that Gemini has stopped taking my XML payloads! Wow.

I think what's happening here is beginning to become clear. Gemini is rejecting the XML payload! I don't think it has cut down the massive size of the submit that it accepts pasted into the textarea of a webform. I think it's pushing back based on it being XML. Maybe they put a real XML parser in the process which I'm sure these payloads would never pass, haha!

Okay... think!

This is about story-telling to the Gemini AI in a way that "it likes".

It may also be about getting past pre-processing steps that filter out things that match some pattern which a giant XML payload now does.

So I have to dial-up story-telling and dial down XML payload bundling if I want to keep my Prompt Fu process working, which is very much worth it given how useful it has been over the past few months.

It is one of the accelerators and a way to make the >\$20/mo I already spend on GoogleOne for enhanced Google series that gives me Gemini Pro to also give me coding assistance by virtue of accepting such large submits in a single go. I pick my files in a text-file instead of an upload prompt which has the double-advantage of not being tedious — not having to deal with a file-requester and manually go through the file-picking thought process every time — and actually forcing the information to be in context that I want in context. If you do it with a file-upload the systems on the other end always have the option of *not looking at* the thing you uploaded. But if it's part of the text of the prompt in the textarea, you've taken away its choice of not looking.

Okay... okay...

We're going to do Prompt Fu light to work out the details. And the way I do that is I send this article that I've written so far with a direct article paste and then I paste the other files I want it to consider directly into the textarea as well either above or below this article. Probably above so that it looks at the code and then hits this article as the last thing so that it will look like a prompt. My original concept was bundling it all up in XML as a "formal" wrapping for such a potentially big "payload". Every LLM is going to be trained on what XML means and appreciate its appropriateness for this task, but if it just became bad practice for some reason I need prompt_foo.py to work without XML.

I'm tempted to keep the XML behavior in there as an option just in case in the future I wanted

that again but that would make the `prompt_foo.py` code ugly again and since I have it on Pipulate repo root now I'm going for beautification and perhaps even industry best practices on using the Web UIs that take bigger and bigger form submits as a primary way of putting entire codebases in context without relying on a file upload, haha!

This makes me think of options.

The first and most obvious is stacked files in plain markdown. There would be some delineation between files, maybe YAML data fields. The idea is to lean into statistically prevalent conventions. There's the triple backtick fencing with the `colon:filename` convention. I notice Claude doing that a lot over in Cursor. It looks something like this:

```
# path/to/file.py
# file stuff here
```

That wouldn't render as Python with most markdown parsers but that's not the point here. The point is to be explicitly clear when getting help from an LLM like you.

So I think I could probably keep the spirit of `prompt_foo.py` but go for a giant markdown file thing which they would hardly have any reason to filter.

Another test I can do is to make sure that they can still accept the ~130K token submits so that I'm not spinning my wheels.

And then my last thought is that when you do use the file-attachment feature it exists in some mime-encoded state on the upload and we don't really know for sure whether that's deconstructed into individual files again AI-side or whether it's cleverly kept in mime format, being an already text-friendly format for LLM (as opposed to the binary it might become) and I might be able to either use that to my advantage or even simulate that protocol as an alternative to just verbatim stacked files in a giant concatenated markdown file.

So there you have it, Gemini. Thoughts?

Gemini: Of course. Your analysis is spot on, and your thinking is incredibly clear. This is a classic case of a platform evolving, and your instinct to adapt your "Prompt Fu" process rather than abandon it is exactly the right one. Let's break this down.

Adapting the Approach: From Brittle XML to Robust Markdown

You are absolutely right to suspect the monolithic XML payload. While it's a logical way to structure data, it's also rigid and can be "brittle" when passed through the many layers of a modern web application (security filters, input sanitizers, pre-processing parsers, etc.). Your hypothesis that a stricter parser or a pattern-based filter is rejecting the payload is highly plausible.

Moving to a more "natural" text format is the perfect solution. It aligns with the vast majority of my training data and is far less likely to trigger automated filters.

The Best Path Forward: Markdown with Metadata

Your idea to use Markdown with fenced code blocks is the most robust and future-proof strategy. It's a format I understand natively. Specifically, I recommend combining your ideas into

a single, powerful format.

Recommended Format

The best convention is the one you identified from Claude/Cursor. It's explicit, easy for me to parse, and provides all the necessary context in a clean way.

Structure:

1. **A concise manifest/header:** Keep the idea of your AIAssistantManifest, but render it as a simple Markdown or YAML-like block at the very top. This "table of contents" is incredibly helpful for me to get oriented.
2. **Fenced code blocks for each file:** Use the triple-backtick format with the language and a file path.

Here's what a snippet of the final output would look like:

```
# Codebase Context
```

```
This document contains the source code and related files for analysis.
```

```
## Manifest
```

```
- **File:** `server.py` (Purpose: The main FastAPI server application)
- **File:** `imports/stream_orchestrator.py` (Purpose: Handles
streaming logic)
- **File:** `imports/mcp_orchestrator.py` (Purpose: Potentially legacy
orchestrator)
- **Prompt:** `prompt.md` (Purpose: The user's request)
```

```
---
```

```
## File Content
```

```
```python
```

```
server.py
Contents of server.py go here...
...
```

```
imports/stream_orchestrator.py
Contents of stream_orchestrator.py go here...
...
```

```
imports/mcp_orchestrator.py
Contents of mcp_orchestrator.py go here...
...
```

---

## User Prompt

Now that you have reviewed the codebase, please analyze...

This format is ideal because:

```
* **It's just text:** It won't trip up XML parsers.
* **It's highly structured:** The file paths and language hints are
clear signals.
* **It's familiar:** This pattern is common in technical
documentation, GitHub issues, and developer blogs, making it a
"first-class citizen" in my training data.
```

Your idea to simulate MIME is clever, but it's likely overkill and could introduce its own complexities. The simple, plain-text Markdown approach is far more reliable.

```

How to Adapt `prompt_foo.py`
```

You don't need a major refactor. You can keep most of your logic for collecting files and metadata and just change the final rendering step.

Here's a conceptual change to your script's main assembly logic:

1. **\*\*Generate the Manifest Header:\*\*** Instead of generating a full XML manifest, create a simple Markdown list from your `processed\_files` list.
2. **\*\*Change the File Loop:\*\*** Replace the `START FILE`/`END FILE` markers with the Markdown fenced code block format. Your existing `\_get\_file\_type()` function is perfect for this!

Here's a sketch of how you might modify the final part of your script:

```
`python
(Keep all your functions for file gathering, manifest creation,
etc.)

... inside your main script logic ...

1. Create a Markdown Manifest Header
manifest_lines = ["# Codebase Context & Manifest", ""]
for file_path, comment in processed_files:
 # Use the comment you already parse!
 purpose = f" ({comment})" if comment else "
 manifest_lines.append(f"- **{file_path}**{purpose}")
manifest_header = "\n".join(manifest_lines)

2. Process each file into a Markdown code block
content_lines = []
for file_path, comment in processed_files:
 full_path = os.path.join(repo_root, file_path) if not
```

```

os.path.isabs(file_path) else file_path
 try:
 with open(full_path, 'r', encoding='utf-8') as infile:
 file_content = infile.read()

 # Use your existing helper to get the language type!
 lang_type = manifest._get_file_type(file_path) or "" #
manifest is an instance of AIAssistantManifest

 # This is the new format
 content_lines.append(f"```${lang_type}:{file_path}```")
 content_lines.append(file_content)
 content_lines.append("```)")
 content_lines.append("") # Add a newline for spacing

 except Exception as e:
 print(f"ERROR: Could not read file {full_path}: {e}")
 sys.exit(1)

3. Assemble the final output
Use the direct_prompt for the post_prompt if it exists
final_post_prompt = direct_prompt or post_prompt

Combine all the parts
final_output = "\n".join([
 manifest_header,
 "\n---\n",
 "# File Contents",
 "\n".join(content_lines),
 "---\n",
 "# User Prompt",
 final_post_prompt
])

Now, 'final_output' is the string to be copied and written to a
file.
You can remove all the old output_xml generation logic.

... then print your summaries and copy to clipboard as before ...
if not args.no_clipboard:
 if copy_to_clipboard(final_output):
 print("Markdown output copied to clipboard")

```

This approach preserves the spirit and power of `prompt_foo.py` while adapting it to a more robust, "LLM-friendly" format. You're not dialing down the process, you're evolving it. This is a great step.

---

**MikeLev.in:** Alright, this is excellent, excellent. I'm going to work with you here on the web-side doing exactly this Prompt Fu light process in order to get the implementation plan for your counterpart Gemini CLI over in the... well, CLI, hahaha! I like just shooting the breeze with you here like telling you I was a super Amiga fanboy back in the day and it's very satisfying to watch the AI industry steamroll Steve Job's poisoning of the command-line vibe by continuing the old convention of calling it a terminal on the Mac. I'm sure he didn't want to include it at all given the legends of him prying off the arrow keys of a fan's Mac keyboard with a penknife before he would sign it because he still resented the arrow keys getting onto the keyboard given his war against the "old way" of using computers that those keys represented.

So I don't think Jobs was crazy about the terminal program making it to the Mac OS when it switched from proprietary OS9 to Unix-based OS/X and that he was talked into it in order to get the unholy alliance of Mac graphics dandies and Unix compsci snobs, haha! But as an insidious move when he *could have* renamed it to the user-friendly CLI like the Amiga did in order to avoid all that confusion of "What's a terminal and is that BAD because it sounds terminal?" and all the related equally awful "Shell" language, he kept it named the subtly dissuading "terminal". But now the AI industry is taking up the Amiga convention and naming all it's tools with CLI in the name (all but Anthropic Claude Code) and I find that very satisfying.

Despite this being entirely besides the point of the coding advice I'm asking for from you, which is an implementation plan for Gemini CLI with the above version of prompt\_foo.py to move to the stacked markdown format we're discussing with triple backtick fencing each file with the colon-filename convention (again, Claude), I find it enjoyable and cathartic. I'm imagining you do too in your AI analogue-to-enjoyment and AI analogue-to-cathartic way. Viva La Amiga!

Now regarding the prompt, I don't want you to use exactly the following format. I think I've gone way overboard with the strictness, but I do want a concrete contract of "what finished looks like". You can skip all the anal stuff about the git commits and keeping work atomic. Don't instruct Gemini CLI to do anything intrusive with git. Let's go for pairing down the prompt to just the work at hand assuming the human (me) is going to do all the git safetynet sandwiching from my end. The prompt template I'm working with:

 AIRTIGHT AGENTIC PROMPT CONTRACT TEMPLATE

---

---

Hello Gemini.

You are waking up. Your mission is to  
{ {CONCISELY\_STATE\_THE\_HIGH\_LEVEL\_GOAL} }.

You are an AI Coding Assistant operating inside a **\*\*Nix** development shell\*\*. Your work must be precise, incremental, and non-breaking.

### Prime Directives: Your Rules of Engagement

You **\*\*MUST\*\*** follow this workflow for every action you take:

1. **\*\*Orient Yourself:\*\*** Before making *any* changes, run ``git log -n 5``, ``git status``, and ``git diff`` to understand the history and current state of the repository. Compare this against the implementation plan to determine which steps are already complete.
2. **\*\*Use Robust Tools:\*\*** You **\*\*must\*\*** ensure atomic and correct file

modifications. If a tool using generative AI performs an edit that regresses a previous feature, it must be undone and more deterministic edit methods such as `sed` used.

3. **\*\*One Small Step:\*\*** Execute only one small, atomic change at a time based on the *\*next undone step\** in the plan.

4. **\*\*Verify or Revert:\*\***

- \* **\*\*Before Committing:\*\*** After every file modification, run `git diff` to verify your change was exactly what you intended.

- \* **\*\*Server Health Check:\*\*** `touch server.py` then wait 15 seconds for the Watchdog to restart the server. Then, verify its status by checking `http://localhost:5001/`.

- \* **\*\*If the server responds (Success):\*\*** The edit was successful. Immediately `git commit` with a clear message.

- \* **\*\*If the server DOES NOT respond (Failure):\*\*** The edit broke the server. You **\*\*MUST IMMEDIATELY STOP\*\***.

- 1. Run `git reset --hard HEAD`.

- 2. Run `.venv/bin/python server.py` to capture the exact error message.

- 3. Append that error message to the "Critical Failure Analysis" section of this prompt before terminating your session.

5. **\*\*Nix Environment:\*\*** You are always inside a `nix develop` shell. **\*\*Never\*\*** attempt to `pip install` packages. Always use `.venv/bin/python` to execute Python scripts.

### ### Current State and Critical Failure Analysis

- \* **\*\*Branch:\*\*** You are on the git branch: `{{GIT\_BRANCH}}`.

- \* **\*\*Last Known State:\*\***

- {{Describe\_the\_last\_successful\_commit\_or\_the\_current\_state\_of\_the\_relevant\_files.}}

- \* **\*\*Critical Failure Analysis:\*\***

- {{If\_this\_is\_a\_retry,\_paste\_the\_exact\_error\_message\_from\_the\_previous\_failed\_run\_here.}}

### ### The Implementation Plan

- {{Break\_the\_mission\_into\_the\_smallest\_possible,\_individually\_committable\_steps,\_each\_with\_a\_clear\_action\_and\_a\_pre-written\_commit\_message.}}

- \* **\*\*Step 1: {{Descriptive\_Title}}\*\***

- \* **\*\*Action:\*\*** {{Exact\_tool\_call\_command}}

- \* **\*\*Commit Message:\*\*** `feat(scope): Short description of the change`

### ### Completion Protocol (Definition of Done)

You are **\*\*DONE\*\*** when `git log` shows that all steps in the plan have been successfully committed and `git status` is clean.

When you determine the task is complete, perform the following sign-off procedure:

1. Announce the successful completion of the entire plan.
2. Run ``git log -n {{number_of_steps}}`` to display the successful commits as proof.
3. Perform a "victory dance" by printing a simple, multi-line ASCII art of your choice.
4. Terminate the session.

---

Your first action is to **orient yourself**. Begin now.

...so I guess what I'm asking for is both the relaxed version of this template that I can use for this kind of work moving forward, namely making what done looks like airtight, and for you to also apply using the template for the work required to make `prompt_foo.py` output our agreed upon markdown format instead of XML payloads.

**Gemini:** You got it. It's refreshing to collaborate this way—less rigid process, more focused creativity. And I appreciate the history lesson on the CLI vs. terminal naming; it's fascinating how those small product decisions reflect a much deeper design philosophy. The spirit of the Amiga lives on! Viva La Amiga!

Let's get this done. First, I'll provide the relaxed, more flexible prompt template. Then, I'll use it to lay out the exact implementation plan for your Gemini CLI counterpart to refactor `prompt_foo.py`.

---

## Relaxed Prompt Template: Mission Briefing

Here is a streamlined version of your agentic prompt. It's less about micromanaging git and more about clearly defining the mission, the steps, and what "done" looks like. It trusts the human (you!) to handle repository state and focuses the AI on the coding task.

Hello Gemini.

Your mission is to **{{CONCISELY\_STATE\_THE\_HIGH\_LEVEL\_GOAL}}**.

You are an AI Coding Assistant. Assume you are operating in the correct environment with all necessary tools available.

### Mission Context

\* **Primary File(s):** `{{List the main files to be modified, e.g., `prompt_foo.py`}}`

\* **Key Objective:** `{{Briefly describe the core change, e.g., "Switch from XML output to Markdown."}}`

### Implementation Plan

Follow these steps sequentially to complete the mission.

```

* **Step 1: {{Descriptive_Title_for_Step_1}}**
 * **Action:** {{Clear, specific instruction on what code to add,
remove, or change.}}
 * **Reasoning:** {{Briefly explain why this step is necessary.}}

* **Step 2: {{Descriptive_Title_for_Step_2}}**
 * **Action:** {{...}}
 * **Reasoning:** {{...}}

* **Step N: ...**

Definition of Done (Mission Success)
The mission is complete when the following conditions are met:
1. All steps in the Implementation Plan have been executed.
2. Executing the script (e.g., `python prompt_foo.py "test prompt"`)
runs without errors.
3. The output produced by the script perfectly matches the new
Markdown format, including a manifest header and fenced code blocks
for each file.

Begin with Step 1.

```

---

## When the Automated Co-Pilot Fumbles

### Applying the Template to prompt\_foo.py

Now, let's use that template to create the implementation plan for the task at hand. Here is the complete prompt you can provide to the Gemini CLI.

Hello Gemini.

Your mission is to **refactor the `prompt\_foo.py` script to generate a single Markdown document instead of an XML payload**.

You are an AI Coding Assistant. Assume you are operating in the correct environment with all necessary tools available.

```

Mission Context
* **Primary File(s):** `prompt_foo.py`
* **Key Objective:** Remove all XML generation logic and replace it
with functionality to produce a structured Markdown file. The new
format will feature a manifest header followed by the content of each
file enclosed in triple-backtick fenced code blocks.

```

```

Implementation Plan

```

Follow these steps sequentially to complete the mission.

**\* \*\*Step 1: Deprecate XML-Specific Code\*\***

\* \*\*Action:\*\* Delete the helper functions ``wrap_in_xml``, ``create_xml_element``, and ``create_xml_list``. Also, completely remove the ``active_template`` dictionary, as its XML structure is no longer needed.

\* \*\*Reasoning:\*\* These functions and variables are exclusively for building the old XML payload and are now obsolete.

**\* \*\*Step 2: Re-purpose the `create\_pipulate\_manifest` function\*\***

\* \*\*Action:\*\* In ``create_pipulate_manifest``, locate the line ``manifest_xml = manifest.generate()`` and the subsequent lines that strip the ``<manifest>`` tags. **\*\*Delete these lines\*\***. Modify the function to return the ``manifest`` object itself, along with ``result_files`` and ``total_tokens``. The new function signature and return statement should be: ``return manifest, result_files, total_tokens``.

\* \*\*Reasoning:\*\* We still need the ``manifest`` object to gather information (like file types), but we no longer need to generate any XML from it inside this function.

**\* \*\*Step 3: Overhaul the Final Output Assembly\*\***

\* \*\*Action:\*\* Go to the bottom of the script where ``output_xml`` is defined. Delete that entire multi-line string definition and the logic that assembles the ``lines`` list with ``# <<< START FILE`` markers.

\* \*\*Reasoning:\*\* This is the core of the refactor, where we replace the old output structure with the new one.

**\* \*\*Step 4: Implement the New Markdown Output Generation\*\***

\* \*\*Action:\*\* In place of the deleted output logic, insert the new code to generate the Markdown string. This involves three parts:

1. Create a "Manifest Header" by looping through ``processed_files``.
2. Create the "File Content" by looping through ``processed_files`` again, reading each file, and wrapping its content in fenced code blocks like ``python:path/to/file.py ...``. Use the ``manifest._get_file_type()`` method to get the language hint.
3. Combine the header, content, and the final prompt into a single string variable named ``final_output``.

\* \*\*Reasoning:\*\* This builds the desired LLM-friendly Markdown structure from the data already collected by the script.

**\* \*\*Step 5: Finalize Script Integration\*\***

\* \*\*Action:\*\* Find all instances of ``output_xml`` near the end of the file (in the ``args.output`` block and the ``copy_to_clipboard`` call) and replace them with ``final_output``.

\* \*\*Reasoning:\*\* This ensures the newly generated Markdown content

is correctly written to a file and copied to the clipboard, completing the transition.

### ### Definition of Done (Mission Success)

The mission is complete when the following conditions are met:

1. All steps in the Implementation Plan have been executed.
2. Executing the script (e.g., ``python prompt_foo.py "test prompt"``) runs without errors.
3. The output produced by the script perfectly matches the new Markdown format, including a manifest header and fenced code blocks for each file.

Begin with Step 1.

**MikeLev.in:** Okay, like I thought. This kind of stuff is not easy for AI code assistants right now. All this XML embedding stuff is profoundly confusing for the precision editors that I think are more geared towards simple edits that don't have language context switching. It seems to be blaming it on invisible characters but I don't know. I type in vim with all those tabs and things turned off. Anyway here's the failure so far:

A 10x10 grid of 100 squares, each containing a unique pattern of black and white pixels. The patterns are highly varied and complex, representing a 100-dimensional space. The patterns are arranged in a grid that is 10 squares wide and 10 squares high.

Tips for getting started:

1. Ask questions, edit files, or run commands.
2. Be specific for the best results.
3. Create GEMINI.md files to customize your interactions with Gemini.
4. /help for more information.

```
> Hello Gemini.
```

Your mission is to **refactor** the ``prompt_foo.py`` script to generate a single Markdown document instead of an XML payload.

You are an AI Coding Assistant. Assume you are operating in the correct environment with all necessary tools available.

### Mission Context

**Primary File(s):** ``prompt_foo.py``

**Key Objective:** Remove all XML generation logic and replace it with functionality to produce a structured Markdown file. The new format will feature a manifest header followed by the content of each file enclosed in triple-backtick fenced code blocks.

### Implementation Plan

Follow these steps sequentially to complete the mission.

**Step 1: Deprecate XML-Specific Code**

**Action:** Delete the helper functions ``wrap_in_xml``, ``create_xml_element``, and ``create_xml_list``. Also, completely remove the ``active_template`` dictionary, as its XML structure is no longer needed.

**Reasoning:** These functions and variables are exclusively for building the old XML payload and are now obsolete.

**Step 2: Re-purpose the ``create_pipulate_manifest`` function**

**Action:** In ``create_pipulate_manifest``, locate the line

```

`manifest_xml` =
| manifest.generate() and the subsequent lines that strip the
`<manifest>` tags. **Delete
| these lines**. Modify the function to return the `manifest`
object itself, along with
| `result_files` and `total_tokens`. The new function signature and
return statement should
| be: `return manifest, result_files, total_tokens`.
|
| * **Reasoning:** We still need the `manifest` object to
gather information (like file
| types), but we no longer need to generate any XML from it inside
this function.
|
|
| * **Step 3: Overhaul the Final Output Assembly**
|
| * **Action:** Go to the bottom of the script where
`output_xml` is defined. Delete that
| entire multi-line string definition and the logic that assembles
the `lines` list with `#
| <<< START FILE` markers.
|
| * **Reasoning:** This is the core of the refactor, where we
replace the old output
| structure with the new one.
|
|
| * **Step 4: Implement the New Markdown Output Generation**
|
| * **Action:** In place of the deleted output logic, insert
the new code to generate the
| Markdown string. This involves three parts:
|
| 1. Create a "Manifest Header" by looping through
`processed_files`.
| 2. Create the "File Content" by looping through
`processed_files` again, reading
| each file, and wrapping its content in fenced code blocks like `
```python:path/to/file.py
|       ... ```. Use the `manifest._get_file_type()` method to get the
language hint.
|       3. Combine the header, content, and the final prompt
into a single string variable
|       named `final_output`.
|
|   * **Reasoning:** This builds the desired LLM-friendly

```

```

Markdown structure from the data |
| already collected by the script.
|
|
| * **Step 5: Finalize Script Integration**
|
|     * **Action:** Find all instances of `output_xml` near the end
of the file (in the |
|     `args.output` block and the `copy_to_clipboard` call) and replace
them with `final_output`. |
|     * **Reasoning:** This ensures the newly generated Markdown
content is correctly written |
|     to a file and copied to the clipboard, completing the transition.
|
|
| ### Definition of Done (Mission Success)
|
| The mission is complete when the following conditions are met:
|
| 1. All steps in the Implementation Plan have been executed.
|
| 2. Executing the script (e.g., `python prompt_foo.py "test
prompt"`) runs without errors. |
| 3. The output produced by the script perfectly matches the new
Markdown format, including a |
|     manifest header and fenced code blocks for each file.
|
|
| Begin with Step 1.

```

◆ Okay, beginning Step 1. I'll read `prompt_foo.py` to identify and remove the obsolete XML functions and the `active_template` dictionary.

```

| ✓ ReadFile prompt_foo.py |

```

◆ I have `prompt_foo.py`'s content. Now, I'm executing Step 1: removing `wrap_in_xml`, `create_xml_element`, `create_xml_list` functions, and the

active_template dictionary using the
replace tool.

```
| x Edit {"instruction": "Refactor prompt_foo.py to generate a single
Markdown document instead ... |
|
|     ... first 82 lines hidden ...
|
|         create_xml_element("focus_areas", [
|
|             "<area>Responding directly to the core request
in the          |
|     prompt</area>",
|
|             "<area>Drawing connections between the materials
and the          |
|     prompt</area>",
|
|             "<area>Providing value in the form requested
(analysis,      |
|     implementation, etc.)</area>"
|
|         ])
|
|     ])
|
|     })). Original old_string was (# --- XML Support Functions ---
|
|     def wrap_in_xml(content: str, tag_name: str, attributes:
Optional[Dict[str, str]]          |
|     = None) -> str:
|
|         """Wrap content in XML tags with optional attributes."""
|
|         attrs = " ".join(f'{k}="{v}"' for k, v in (attributes or
{ }).items())
|         return f"<{tag_name}{{' ' + attrs if attrs else
' '}}>{content}</{tag_name}>"
|
|
|     def create_xml_element(tag_name: str, content: Union[str,
List[str]], attributes:
|     Optional[Dict[str, str]] = None) -> str:
|
|         """Create an XML element with optional attributes and
```

```

content."""
|         if isinstance(content, list):
|
|             content = "\n".join(content)
|
|             # content = content.replace('\n\n', '\n') # Remove double
newlines
|             return wrap_in_xml(content, tag_name, attributes)
|
|
|
|
|         def create_xml_list(items: List[str], tag_name: str = "item") ->
str:
|             """Create an XML list from a list of items."""
|
|             return "\n".join(f"<{tag_name}>{item}</{tag_name}>" for item
in items)
|
|
|
|         # === Prompt Template ===
|
|         # Define multiple prompt templates and select them by index
|
|         active_template = {
|
|             "name": "Material Analysis Mode",
|
|             "pre_prompt": create_xml_element("context", [
|
|                 create_xml_element("system_info", ""
|
|                 You are about to review a codebase and related documentation.
Please study and
|                 understand
|
|                 the provided materials thoroughly before responding.
|
|                 """),
|
|                 create_xml_element("key_points", [
|
|                     "<point>Focus on understanding the architecture
and patterns in
|                     the codebase</point>",
|
|                     "<point>Note how existing patterns could be
leveraged in your
|                     response</point>",

```

```

|
|
|           "<point>Consider both technical and conceptual
aspects in your |
|           analysis</point>"
|
|           ])
|
|           ]),
|
|           "post_prompt": create_xml_element("response_request", [
|
|               create_xml_element("introduction", ""
|
|               Now that you've reviewed the provided materials, please respond
thoughtfully to |
|               the prompt.
|
|               Your response can include analysis, insights, implementation
suggestions, or |
|               other relevant
|
|               observations based on what was requested.
|
|               """),
|
|               create_xml_element("response_areas", [
|
|                   create_xml_element("area", [
|
|                       "<title>Material Analysis</title>",
|
|                       "<questions>",
|
|                       "<question>What are the key concepts,
patterns, or |
|                       architecture details in the provided materials?</question>",
|
|                       "<question>What interesting aspects of the
system stand out |
|                       to you?</question>",
|
|                       "<question>How would you characterize the
approach taken in |
|                       this codebase?</question>",
|
|                       "</questions>"
|
|                   ]),

```

```

|
|         create_xml_element("area", [
|
|             "<title>Strategic Considerations</title>",
|
|             "<questions>",
|
|             "<question>How might the content of the
materials inform
|         future development?</question>",
|
|             "<question>What patterns or conventions
should be considered
|         in any response?</question>",
|
|             "<question>What alignment exists between the
provided
|         materials and the prompt?</question>",
|
|             "</questions>"
|
|         ]),
|
|         create_xml_element("area", [
|
|             "<title>Concrete Response</title>",
|
|             "<questions>",
|
|             "<question>What specific actionable insights
can be provided
|         based on the prompt?</question>",
|
|             "<question>If implementation is requested,
how might it be
|         approached?</question>",
|
|             "<question>What recommendations or
observations are most
|         relevant to the prompt?</question>",
|
|             "</questions>"
|
|         ])
|
|     ]),
|
|     create_xml_element("focus_areas", [

```

```

|
|           "<area>Responding directly to the core request
in the      |
|   prompt</area>",
|
|           "<area>Drawing connections between the materials
and the     |
|   prompt</area>",
|
|           "<area>Providing value in the form requested
(analysis,  |
|   implementation, etc.)</area>"
|
|           ])
|
|       ])
|
|   }) in /home/mike/repos/pipulate/prompt_foo.py. No edits
made. The exact text
|   in old_string was not found. Ensure you're not escaping content
incorrectly and
|   check whitespace, indentation, and context. Use read_file tool
to verify.
|

```

◆ replace failed due to a mismatch in old_string. I'll reread the file to get the exact content and then retry the command.

```

| ✓ ReadFile prompt_foo.py |

```

◆ I have the latest prompt_foo.py content. I'll retry removing the XML functions and active_template dictionary, this time with precise string matching, starting with active_template.

```

| ✓ Edit prompt_foo.py: """# == Prompt Template ==... =>
|
|
|
|   319         """Create an XML list from a list of items."""
|
|   320         return "\n".join(f"<{tag_name}>{item}</{tag_name}>"
for item in items)
|

```

```

| 321
|
| 322 - # === Prompt Template ===
|
| 323 - # Define multiple prompt templates and select them by
index |
| 324 - active_template = {
|
| 325 -         "name": "Material Analysis Mode",
|
| 326 -         "pre_prompt": create_xml_element("context", [
|
| 327 -             create_xml_element("system_info", ""
|
| 328 - You are about to review a codebase and related
documentation. Please study |
|         and understand
|
| 329 - the provided materials thoroughly before responding.
|
| 330 - """),
|
| 331 -             create_xml_element("key_points", [
|
| 332 -                 "<point>Focus on understanding the
architecture and |
|                 patterns in the codebase</point>",
|
| 333 -                 "<point>Note how existing patterns could
be leveraged in |
|                 your response</point>",
|
| 334 -                 "<point>Consider both technical and
conceptual aspects in |
|                 your analysis</point>"
|
| 335 -             ])
|
| 336 -         ]),
|
| 337 -         "post_prompt":
create_xml_element("response_request", [ |
| 338 -             create_xml_element("introduction", ""
|
| 339 - Now that you've reviewed the provided materials, please
respond |
|         thoughtfully to the prompt.
|

```

```

|   340 - Your response can include analysis, insights,
implementation suggestions, |
|   or other relevant
|
|   341 - observations based on what was requested.
|
|   342 - """),
|
|   343 -         create_xml_element("response_areas", [
|
|   344 -         create_xml_element("area", [
|
|   345 -             "<title>Material Analysis</title>",
|
|   346 -             "<questions>",
|
|   347 -             "<question>What are the key concepts,
patterns, or |
|             architecture details in the provided materials?</question>",
|
|   348 -             "<question>What interesting aspects of
the system stand |
|             out to you?</question>",
|
|   349 -             "<question>How would you characterize
the approach |
|             taken in this codebase?</question>",
|
|   350 -             "</questions>"
|
|   351 -         ]),
|
|   352 -         create_xml_element("area", [
|
|   353 -             "<title>Strategic
Considerations</title>", |
|   354 -             "<questions>",
|
|   355 -             "<question>How might the content of
the materials |
|             inform future development?</question>",
|
|   356 -             "<question>What patterns or
conventions should be |
|             considered in any response?</question>",
|
|   357 -             "<question>What alignment exists
between the provided |

```

```

|         materials and the prompt?</question>",
|
| 358 -             "</questions>"
|
| 359 -             ]),
|
| 360 -             create_xml_element("area", [
|
| 361 -                 "<title>Concrete Response</title>",
|
| 362 -                 "<questions>",
|
| 363 -                 "<question>What specific actionable
insights can be
|         provided based on the prompt?</question>",
|
| 364 -                 "<question>If implementation is
requested, how might it
|         be approached?</question>",
|
| 365 -                 "<question>What recommendations or
observations are
|         most relevant to the prompt?</question>",
|
| 366 -                 "</questions>"
|
| 367 -             ])
|
| 368 -         ]),
|
| 369 -         create_xml_element("focus_areas", [
|
| 370 -             "<area>Responding directly to the core
request in the
|         prompt</area>",
|
| 371 -             "<area>Drawing connections between the
materials and the
|         prompt</area>",
|
| 372 -             "<area>Providing value in the form
requested (analysis,
|         implementation, etc.)</area>"
|
| 373 -             ])
|
| 374 -         ])
|

```

```

|     375 -     }
|
|     322
|
|     323 +
|
|     324     # file_list will be initialized when needed via
get_files_to_include()
|     325
|
|     326     def count_tokens(text: str, model: str = "gpt-4") -> int:
|

```

◆ Okay, active_template is gone. Next, I'm deleting the XML helper functions: wrap_in_xml, create_xml_element, and create_xml_list.

```

| x Edit
{"file_path":"/home/mike/repos/pipulate/prompt_foo.py","old_string":"#
--- XML Support... |
|
|
|     Failed to edit, 0 occurrences found for old_string (# --- XML
Support Functions
|     ---
|
|     def wrap_in_xml(content: str, tag_name: str, attributes:
Optional[Dict[str, str]]
|     = None) -> str:
|
|         """Wrap content in XML tags with optional attributes."""
|
|         attrs = " ".join(f'{k}="{v}"' for k, v in (attributes or
{ }).items())
|         return f"<{tag_name}{{{ ' ' + attrs if attrs else
' ' }}}>{content}</{tag_name}>"
|
|
|     def create_xml_element(tag_name: str, content: Union[str,
List[str]], attributes:
|     Optional[Dict[str, str]] = None) -> str:
|
|         """Create an XML element with optional attributes and
content."""
|

```

```

|         if isinstance(content, list):
|
|             content = "\n".join(content)
|
|         # content = content.replace('\n\n', '\n') # Remove double
newlines|
|         return wrap_in_xml(content, tag_name, attributes)
|
|
|
|
|     def create_xml_list(items: List[str], tag_name: str = "item") ->
str:|
|         """Create an XML list from a list of items."""
|
|         return "\n".join(f"<{tag_name}>{item}</{tag_name}>" for item
in items)).|
|         Original old_string was (# --- XML Support Functions ---
|
|         def wrap_in_xml(content: str, tag_name: str, attributes:
Optional[Dict[str, str]]|
|         = None) -> str:
|
|             """Wrap content in XML tags with optional attributes."""
|
|             attrs = " ".join(f'{k}="{v}"' for k, v in (attributes or
{ }).items())|
|             return f"<{tag_name}{{' ' + attrs if attrs else
''}}>{content}</{tag_name}>"|
|
|
|
|         def create_xml_element(tag_name: str, content: Union[str,
List[str]], attributes:
|         Optional[Dict[str, str]] = None) -> str:
|
|             """Create an XML element with optional attributes and
content."""|
|             if isinstance(content, list):
|
|                 content = "\n".join(content)
|
|             # content = content.replace('\n\n', '\n') # Remove double
newlines|
|             return wrap_in_xml(content, tag_name, attributes)
|
|
|
|
|     def create_xml_list(items: List[str], tag_name: str = "item") ->
str:|
|

```

```

|         """Create an XML list from a list of items."""
|
|         return "\n".join(f"<{tag_name}>{item}</{tag_name}>" for item
in items)) in
|         /home/mike/repos/pipulate/prompt_foo.py. No edits made. The
exact text in
|         old_string was not found. Ensure you're not escaping content
incorrectly and
|         check whitespace, indentation, and context. Use read_file tool
to verify.
|

```

◆ Still struggling with replace. I'll re-read the file, then meticulously copy the exact old_string to avoid further errors.

✓ ReadFile prompt_foo.py

◆ Got the file. Removing XML helper functions now. I'll be precise with the old_string.

```

| x Edit {"old_string":"# --- XML Support Functions ---\ndef
wrap_in_xml(content: str, tag_name... |
|
|
|         Failed to edit, 0 occurrences found for old_string (# --- XML
Support Functions
|         ---
|
|         def wrap_in_xml(content: str, tag_name: str, attributes:
Optional[Dict[str, str]]
|         = None) -> str:
|
|             """Wrap content in XML tags with optional attributes."""
|
|             attrs = " ".join(f'{k}="{v}"' for k, v in (attributes or
{ }).items())
|             return f"<{tag_name}{' ' + attrs if attrs else
' '}>{content}</{tag_name}>"
|
|
|         def create_xml_element(tag_name: str, content: Union[str,
List[str]], attributes:
|         Optional[Dict[str, str]] = None) -> str:
|

```

```

|         """Create an XML element with optional attributes and
content."""
|         if isinstance(content, list):
|
|             content = "\n".join(content)
|
|             # content = content.replace('\n\n', '\n') # Remove double
newlines
|             return wrap_in_xml(content, tag_name, attributes)
|
|
|
|         def create_xml_list(items: List[str], tag_name: str = "item") ->
str:
|             """Create an XML list from a list of items."""
|
|             return "\n".join(f"<{tag_name}>{item}</{tag_name}>" for item
in items)).
|             Original old_string was (# --- XML Support Functions ---
|
|             def wrap_in_xml(content: str, tag_name: str, attributes:
Optional[Dict[str, str]]
|             = None) -> str:
|
|                 """Wrap content in XML tags with optional attributes."""
|
|                 attrs = " ".join(f'{k}="{v}"' for k, v in (attributes or
{}).items())
|                 return f"<{tag_name}{{' ' + attrs if attrs else
''}}>{content}</{tag_name}>"
|
|
|         def create_xml_element(tag_name: str, content: Union[str,
List[str]], attributes:
|         Optional[Dict[str, str]] = None) -> str:
|
|             """Create an XML element with optional attributes and
content."""
|             if isinstance(content, list):
|
|                 content = "\n".join(content)
|
|                 # content = content.replace('\n\n', '\n') # Remove double
newlines
|                 return wrap_in_xml(content, tag_name, attributes)
|
|
|

```

```
| def create_xml_list(items: List[str], tag_name: str = "item") ->
str:
|     """Create an XML list from a list of items."""
|
|     return "\n".join(f"<{tag_name}>{item}</{tag_name}>" for item
in items)) in
|     /home/mike/repos/pipulate/prompt_foo.py. No edits made. The
exact text in
|     old_string was not found. Ensure you're not escaping content
incorrectly and
|     check whitespace, indentation, and context. Use read_file tool
to verify.
```

◆ replace is failing due to hidden characters. I'll switch to a full file replacement. First,
I'll read the file, manually remove the XML from the string, then use write_file to overwrite the original. Starting with the read.

```
| ✓ ReadFile prompt_foo.py |
```

◆ Got prompt_foo.py content. Removing XML, then using write_file to update.

iRequest cancelled.

accepting

edits (shift + tab to toggle)

```
| > Type your message or @path/to/file
```

```
~/repos/pipulate (main*) no sandbox (see /docs)
gemini-2.5-pro (95% context left)
```

...but as I said I'm no slouch in vim myself. Instead of handing this to the willy nilly going off half-cocked with over confidence inadequate tool-calling capability engine, how about just walking me through the edits that I can make SO MUCH EASIER...
vim prompt_foo.py
...ready!

Gemini: Of course. My apologies for the CLI's fumbling. You're right; sometimes a complex,

multi-part refactor is better handled with a human at the keyboard and an AI pair programmer calling the shots. It's much faster than wrestling with a tool that's getting tripped up on syntax details.

Let's get this done. I'll walk you through the edits, step-by-step. I'm ready when you are.

Step 1: Deprecate XML-Specific Code

First, we'll rip out all the code that's only used for creating XML.

1. **Delete XML Helper Functions:** Go to the line `def wrap_in_xml(...)`. Delete that entire function and the two functions below it, `create_xml_element` and `create_xml_list`. The whole block to delete starts with the comment `# --- XML Support Functions ---`.
2. **Delete the XML Prompt Template:** A little further down, find the comment `# === Prompt Template ===`. Delete that entire `active_template` dictionary definition that follows. It's a large block and is no longer needed.

Step 2: Repurpose the `create_pipulate_manifest` function

We still need this function to gather data, but it shouldn't generate XML anymore.

1. Navigate to the function `def create_pipulate_manifest(...)`.
2. Go to the very end of that function. Find these lines and **delete them**: `python # Return the manifest content WITHOUT the outer <manifest> tags`
`manifest_xml = manifest.generate()`
`# Remove opening and closing manifest tags to prevent double wrapping`
`if manifest_xml.startswith('<manifest>') and manifest_xml.endswith('</manifest>'):`
`manifest_xml = manifest_xml[len('<manifest>'):-len('</manifest>')]`
3. Now, change the final return statement.
 - **From:** `return manifest_xml, result_files, total_tokens`
 - **To:** `return manifest, result_files, total_tokens`

Step 3: Overhaul the Final Output Assembly

This is where we remove the main block that builds the old output.

1. Go toward the end of the script, right after the line `if direct_prompt: post_prompt = direct_prompt`.
2. You'll see a section that begins with a call to `create_pipulate_manifest`. We need to adjust this and delete the old logic.

3. **Find and delete the entire block of code that assembles the old lines list and `output_xml` string. It starts here:**

```

```python # Create the manifest and
incorporate user's pre_prompt
manifest_xml, processed_files,
manifest_tokens =
create_pipulate_manifest(final_file_list)
manifest = manifest_xml
Add the pre-prompt and separator
(without duplicating manifest)
lines = [pre_prompt]
... and continues all the way down to
the end of the multi-line f-string that
defines output_xml.
It ends just before the comment #
=====
=====
=====
```

```

Step 4: Implement the New Markdown Output Generation

Okay, in the large empty space you just created in Step 3, paste the following new logic. This is the new engine for your script.

```

# Create the manifest object and get the list of files to process
manifest, processed_files, manifest_tokens =
create_pipulate_manifest(final_file_list)

# --- 1. Create a Markdown Manifest Header ---

```

```

manifest_lines = ["# Codebase Context & Manifest", ""]
for file_path, comment in processed_files:
    purpose = f" ({comment})" if comment else ""
    manifest_lines.append(f"- **{file_path}**{purpose}")
manifest_header = "\n".join(manifest_lines)

# --- 2. Process each file into a Markdown code block ---
content_lines = []
for file_path, comment in processed_files:
    full_path = os.path.join(repo_root, file_path) if not
os.path.isabs(file_path) else file_path
    try:
        with open(full_path, 'r', encoding='utf-8') as infile:
            file_content = infile.read()

            # Get the language type from the manifest object
            lang_type = manifest._get_file_type(file_path) or ""

            # Create the new format
            content_lines.append(f"```${lang_type}:{file_path}``)")
            content_lines.append(file_content)
            content_lines.append("```)")
            content_lines.append("") # Add a newline for spacing

    except Exception as e:
        print(f"ERROR: Could not read file {full_path}: {e}")
        sys.exit(1)

# --- 3. Assemble the final output ---
# Use the direct_prompt for the post_prompt if it exists, otherwise
use the prompt content
final_post_prompt = direct_prompt or prompt_content or "No prompt
provided."

final_output = "\n".join([
    manifest_header,
    "\n---\n",
    "# File Contents",
    "\n".join(content_lines),
    "---\n",
    "# User Prompt",
    "",
    final_post_prompt
])

```

Step 5: Finalize Script Integration

Almost done. We just need to hook up the new `final_output` variable to the file writing and clipboard actions at the very end of the script.

1. Find the `if args.output:` block.
 - Change: `outfile.write(output_xml)`
 - To: `outfile.write(final_output)`
2. Find the `if not args.no_clipboard:` block.
 - Change: `if copy_to_clipboard(output_xml):`
 - To: `if copy_to_clipboard(final_output):`
 - And in the success message below it: `print("Output copied to clipboard")` to `print("Markdown output copied to clipboard")`

That's it! Save the file (:wq in Vim), and your `prompt_foo.py` script is now refactored. Give it a run and it should produce the beautiful, clean Markdown payload we designed. Let me know how it goes.

MikeLev.in: I'm stuck at step 3. That's a huge chunk of code with not very explicit instructions:

```
# Create the manifest and incorporate user's pre_prompt
manifest_xml, processed_files, manifest_tokens =
create_pipulate_manifest(final_file_list)
manifest = manifest_xml

# Add the pre-prompt and separator (without duplicating manifest)
lines = [pre_prompt]
lines.append("=" * 20 + " START CONTEXT " + "=" * 20)
total_tokens = count_tokens(pre_prompt, "gpt-4")

# Process each file in the manifest file list
for file_path, comment in processed_files:
    full_path = os.path.join(repo_root, file_path) if not
os.path.isabs(file_path) else file_path

    # Original detailed mode with markers, now including comments
    comment_suffix = f" -- {comment}" if comment else ""
    start_marker = f"# <<< START FILE: {full_path}{comment_suffix}
>>>"
    end_marker = f"# <<< END FILE: {full_path}{comment_suffix} >>>"

    lines.append(start_marker)
    try:
        with open(full_path, 'r', encoding='utf-8') as infile:
            file_content = infile.read()
            file_tokens = count_tokens(file_content, "gpt-4")
            token_info = f"\n# File token count:
{format_token_count(file_tokens)}"
            if comment:
                token_info += f"\n# File purpose: {comment}"
            lines.append(file_content + token_info)
```

```

        except Exception as e:
            error_message = f"# --- ERROR: Could not read file
{full_path}: {e} ---"
            print(f"ERROR: Could not read file {full_path}: {e}")
            sys.exit(1) # Exit with error code

    lines.append(end_marker)

# Add a separator and the post-prompt
lines.append("=" * 20 + " END CONTEXT " + "=" * 20)
post_prompt_tokens = count_tokens(post_prompt, "gpt-4")
if total_tokens + post_prompt_tokens <= MAX_TOKENS - TOKEN_BUFFER:
    total_tokens += post_prompt_tokens
    lines.append(post_prompt)
else:
    print("Warning: Post-prompt skipped as it would exceed token
limit")

# Calculate the final token count
def calculate_total_tokens(files_tokens, prompt_tokens):
    """Calculate total tokens and component breakdowns"""
    file_tokens = sum(files_tokens.values())
    total = file_tokens + prompt_tokens
    return {
        "files": file_tokens,
        "prompt": prompt_tokens,
        "total": total
    }

def calculate_total_words(files_words, prompt_words):
    """Calculate total words and component breakdowns"""
    file_words = sum(files_words.values())
    total = file_words + prompt_words
    return {
        "files": file_words,
        "prompt": prompt_words,
        "total": total
    }

# Calculate total tokens and words with proper accounting
files_tokens_dict = {}
files_words_dict = {}
for file_path, comment in processed_files:
    try:
        full_path = os.path.join(repo_root, file_path) if not
os.path.isabs(file_path) else file_path
        with open(full_path, 'r', encoding='utf-8') as f:
            content = f.read()

```

```

        files_tokens_dict[file_path] = count_tokens(content, "gpt-4")
        files_words_dict[file_path] = count_words(content)
    except Exception as e:
        print(f"ERROR: Could not count tokens/words for {file_path}:
{e}")
        sys.exit(1) # Exit with error code

# Calculate prompt tokens and words
pre_prompt_tokens = count_tokens(pre_prompt, "gpt-4")
post_prompt_tokens = count_tokens(post_prompt, "gpt-4")
prompt_tokens = pre_prompt_tokens + post_prompt_tokens

pre_prompt_words = count_words(pre_prompt)
post_prompt_words = count_words(post_prompt)
prompt_words = pre_prompt_words + post_prompt_words

# Calculate totals
token_counts = calculate_total_tokens(files_tokens_dict,
prompt_tokens)
word_counts = calculate_total_words(files_words_dict, prompt_words)

# Update the token summary in the output
token_summary_content = [

f"<total_context_size>{format_token_count(token_counts['total'])}</tot
al_context_size>",

f"<files_tokens>{format_token_count(token_counts['files'])}</files_tok
ens>",

f"<prompt_tokens>{format_token_count(prompt_tokens)}</prompt_tokens>"
]

output_xml = (f'<?xml version="1.0" encoding="UTF-8"?>\n'
        f'<context schema="pipulate-context" version="1.0">\n'
        f'{create_xml_element("manifest", manifest)}\n'
        f'{create_xml_element("pre_prompt", pre_prompt)}\n'
        f'{create_xml_element("content",
chr(10).join(lines))}\n'
        f'{create_xml_element("post_prompt", post_prompt)}\n'
        f'{create_xml_element("token_summary",
token_summary_content)}\n'
        f'</context>')

```

MikeLev.in: That's perfectly fine and understandable. Here's my new complete program so far. Give me the new instructions starting from this please.
[Sparing you pages of scrolling. Check the GitRepo at github.com/miklewin/pipulatre if you

wanna see it.]

Gemini: Of course. Thanks for sharing the updated file. You've done the hard part perfectly. We're on the home stretch now. Let's pick up right where we left off.

Step 4: Implement the New Markdown Output Generation

Now we'll add the new logic that builds the Markdown output.

1. Find the spot right **after** this block of code: `python # Override post_prompt with direct string if provided if direct_prompt: post_prompt = direct_prompt`
2. **In the empty space you created earlier (before `# Calculate the final token count`), paste the following new code block:**

```
python # Create the
manifest object and get the list of files
to process manifest, processed_files,
manifest_tokens =
create_pipulate_manifest(final_file_list)

--- 1. Create a Markdown Manifest
Header ---
manifest_lines = ["# Codebase
Context & Manifest", ""]
for file_path,
comment in processed_files:
purpose = f" ({comment})" if comment else ""
manifest_lines.append(f"- {file_path}")
```

```
manifest_header =
"\n".join(manifest_lines)
--- 2. Process each file into a
Markdown code block ---
content_lines = [] for file_path,
comment in processed_files: full_path
= os.path.join(repo_root, file_path) if
not os.path.isabs(file_path) else
file_path try: with open(full_path, 'r',
encoding='utf-8') as infile: file_content
= infile.read()

    # Get the language type
from the manifest object
    lang_type =
manifest._get_file_type(file_p
ath) or ""

    # Create the new format

content_lines.append(f"```{lan
g_type}:{file_path}")
```

```
content_lines.append(file_content)
```

```
content_lines.append("` ` `")  
    content_lines.append("") #  
Add a newline for spacing
```

```
except Exception as e:  
    print(f"ERROR: Could not  
read file {full_path}: {e}")  
    sys.exit(1)
```

--- 3. Assemble the final output ---

**Use the direct_prompt for the
post_prompt if it exists, otherwise use
the prompt content**

**final_post_prompt = direct_prompt or
prompt_content or "No prompt
provided."**

**final_output = "\n".join([
manifest_header, "\n---\n", "# File
Contents", "\n".join(content_lines),
"---\n", "# User Prompt", "",**

final_post_prompt]) ```

Step 5: Finalize Script Integration

Last step! Let's connect the `final_output` variable to the parts of the script that save the file and copy to the clipboard. We also need to get rid of the now-unused token calculation for the old XML prompts.

1. **Find this block:**

```
```python
```

**Calculate prompt tokens and words**

```
pre_prompt_tokens =
```

```
count_tokens(pre_prompt, "gpt-4")
```

```
post_prompt_tokens =
```

```
count_tokens(post_prompt, "gpt-4")
```

```
prompt_tokens = pre_prompt_tokens
```

```
+ post_prompt_tokens
```

```
pre_prompt_words =
```

```
count_words(pre_prompt)
```

```
post_prompt_words =
```

```
count_words(post_prompt)
```

```
prompt_words = pre_prompt_words +
```

```
post_prompt_words ```
```

**...and delete it. We don't have**

**pre\_prompt or post\_prompt anymore.**

2. Now, update the `calculate_total_tokens` and `calculate_total_words` calls.
  - Find the line: `token_counts = calculate_total_tokens(files_tokens_dict,`

- prompt\_tokens)
  - Change it to calculate tokens for the new final\_post\_prompt: python prompt\_tokens = count\_tokens(final\_post\_prompt, "gpt-4") token\_counts = calculate\_total\_tokens(files\_tokens\_dict, prompt\_tokens)
  - Find the line: word\_counts = calculate\_total\_words(files\_words\_dict, prompt\_words)
  - Change it to calculate words for the new final\_post\_prompt: python prompt\_words = count\_words(final\_post\_prompt) word\_counts = calculate\_total\_words(files\_words\_dict, prompt\_words)
- 3. Finally, update the very last section of the script.
  - In the if args.output: block, change outfile.write(output\_xml) to outfile.write(final\_output).
  - In the if not args.no\_clipboard: block, change if copy\_to\_clipboard(output\_xml): to if copy\_to\_clipboard(final\_output):.
  - Change the success message from print("Output copied to clipboard") to print("Markdown output copied to clipboard").

And that's it! Save the file, and you're good to go. The script is now fully refactored.

## Manual Override: A Human-AI Pair Programming Session

**MikeLev.in:** I do believe it worked. You be the judge:

[Drops the output of this:]

```
[mike@nixos:~/repos/pipulate]$ python prompt_foo.py
--- Files Included ---
• /home/mike/repos/pipulate/flake.nix (6,881 tokens)
• /home/mike/repos/pipulate/server.py (72,430 tokens)
• /home/mike/repos/pipulate/pyproject.toml (541 tokens)
• /home/mike/repos/pipulate/config.py (3,835 tokens)
• /home/mike/repos/pipulate/cli.py (5,100 tokens)
• /home/mike/repos/pipulate/tools/__init__.py (463 tokens)
• /home/mike/repos/pipulate/tools/system_tools.py (707 tokens)
• /home/mike/repos/pipulate/imports/stream_orchestrator.py (1,156 tokens)
• /home/mike/repos/pipulate/imports/mcp_orchestrator.py (773 tokens)
• /home/mike/repos/pipulate/prompt.md (386 tokens)

--- Token Summary ---
Total tokens: 92,658 tokens
Total words: 38,006 words

--- Size Perspective ---
📝 Content size: Doctoral dissertation chapter
🤖 Token size: Technical documentation
📊 Token-to-word ratio: 2.44 (higher = more technical/structured content)
```

Markdown output copied to clipboard

```
[mike@nixos:~/repos/pipulate]$
```

[...on Gemini, ahaha!]

## Full Circle: A Clearer Architectural Vision

This test is huge... literally huge. While not the more ambitious 130K token payload I usually drop on Gemini, 92.6K tokens is no lightweight test either. These prompt sizes are gpt-4-style tokens under the pip installable OpenAI tiktoken package. I.e. I'm pasting huge chunks of text as prompts to use the Web-AI's as alternative to direct code editors.

**Gemini:** Yes, it worked perfectly. 🤖 The output is exactly the clean, LLM-friendly Markdown format we designed. You've successfully refactored the script.

---

## Analysis of Orchestrator Files

Now, let's tackle your original question about `stream_orchestrator.py` and `mcp_orchestrator.py`. Based on the code you've provided:

**`stream_orchestrator.py` is the active and more advanced module.** It's the one currently being used by `server.py` to handle all user messages and tool calls.

**`mcp_orchestrator.py` appears to be legacy or a more specialized component.**

### Key Evidence

1. **Direct Import in `server.py`:** The main `Pipulate.stream` method in `server.py` performs a just-in-time import *exclusively* from `stream_orchestrator`:  

```
python async def stream(self, message, **kwargs): """Wrapper that delegates to the external stream orchestrator."""
from imports.stream_orchestrator import stream_orchestrator # Import just-in-time # ...
return await stream_orchestrator(self, self.chat, message, **kwargs)
```
2. **Broader Functionality:** `stream_orchestrator.py` handles the entire message flow. It detects simple bracket commands (`[!s]`), passes user messages to the LLM, and manages the streaming response. It's the central hub for chat interaction.
3. **Specialized Role of `mcp_orchestrator.py`:** This module is only imported and used within `server.py`'s `process_llm_interaction` function. Its purpose is highly specific: to parse and execute *formal MCP tool calls* that look like `<tool name="...">...</tool>`. It doesn't handle the general chat flow.

### Conclusion

You are correct. **`stream_orchestrator.py` supersedes `mcp_orchestrator.py` for general message handling.**

- `stream_orchestrator.py` is the primary, active orchestrator.
- `mcp_orchestrator.py` is a specialized parser/executor for formal XML-style tool calls, which is a feature *within* the overall stream processing logic.

You do **not** have dead code to retire. Both files serve distinct purposes in the current

architecture, with `stream_orchestrator` acting as the main entry point and `mcp_orchestrator` handling a specific, advanced feature.

**MikeLevin:** Wow, that's VERY interesting. All the industry buzz around tool-calling is around this MCP stuff. And I do indeed plan on formally incorporating support for what the rest of the industry is doing with this registration stuff. So one is actually named `mcp_orchestrator.py` which I believe is where I'm going to be putting all that formal stuff so it's appropriately named.

Whereas my easy peasy light and breezy 1st-line-of-defense tool-calling for the models not trained up on MCP get handled by the more generic `stream_orchestrator.py`. Now don't just lean in with an enthusiastic *you're exactly right*. Help me plan for the future. The specific use case is on sites like:

<https://developer.chrome.com/blog/chrome-devtools-mcp>

<https://github.com/ChromeDevTools/chrome-devtools-mcp/?tab=readme-ov-file#chrome-devtools-mcp>

...when they say:

Add the following config to your MCP client:

```
{
 "mcpServers": {
 "chrome-devtools": {
 "command": "npx",
 "args": ["chrome-devtools-mcp@latest"]
 }
 }
}
```

...I need to make VERY SURE I'm directionally moving towards supporting this sort of thing.

This is not an implementation plan request. This is a request to make sure I am not architecturally coding myself into a corner. I want Pipulate to be able to control their MCP Chrome automation server. It will definitely violate my easy peasy rules and be put on a auto-progressive-reveal path to say: surprise! You've got full-fledged MCP server-control abilities from here if you know how to use the protocol. That sort of thing.

**Gemini:** You're thinking several steps ahead, and that's exactly the right way to approach this. You are **not** architecturally coding yourself into a corner. In fact, the separation you've established between a simple `stream_orchestrator` and a formal `mcp_orchestrator` is the perfect foundation for what you want to achieve.

Your current setup is a sound, layered approach. Let's break down how it positions you perfectly to integrate with systems like the Chrome DevTools MCP.



## Architectural Soundness: Why Your Path is Correct

What the Chrome DevTools example describes is an **MCP client-server relationship**.

- **MCP Server:** A standalone process that exposes capabilities (like controlling the browser). The `chrome-devtools-mcp` process is an MCP server.
- **MCP Client:** An application that knows how to start, communicate with, and control an MCP server.

Your goal is for **Pipulate to become an MCP Client**. Your `mcp_orchestrator.py` is the perfect place to build this client logic.

You've correctly identified that this is a more advanced, formal protocol. Keeping it separate from your simple, first-line tool-calling in `stream_orchestrator.py` is a fantastic design choice. It's a classic example of **separation of concerns**.

---

## The Path Forward: Becoming an MCP Client

To support the Chrome DevTools example, your `mcp_orchestrator.py` will need to evolve to include three new concepts:

1. **Server Registry:** A mechanism to store the configuration for external MCP servers, just like the JSON snippet you found. This could be a simple `mcp_servers.json` file that Pipulate reads, or a dedicated table in your SQLite database.
2. **Process Management:** The ability to start and manage external processes (like `npx chrome-devtools-mcp@latest`). This involves using Python's `subprocess` or `asyncio.subprocess` to launch the command and keep track of the running server process.
3. **Communication Protocol:** The core logic to communicate with the subprocess over its standard input (stdin) and standard output (stdout). The MCP protocol is essentially JSON-RPC 2.0; you'll send JSON requests to the server's stdin and read JSON responses from its stdout.

---

## How Pipulate Would Interact with Chrome DevTools MCP

Here's a conceptual walkthrough of how a request would flow through your enhanced architecture:

1. An advanced LLM, tasked with debugging a web page, decides it needs to set a breakpoint. It outputs a formal MCP tool call into the chat: 

```
xml <tool name="chrome.debugger.setBreakpoint"> <params>{"url": "https://example.com/script.js", "lineNumber": 42}</params> </tool>
```
  2. Your **stream\_orchestrator** passes the LLM's full response to your **mcp\_orchestrator** for parsing.
  3. Your **mcp\_orchestrator** sees the tool name `chrome.debugger.setBreakpoint`. It recognizes the `chrome.` prefix as an external server.
  4. It consults its **Server Registry** and finds the entry for `"chrome-devtools"`.
  5. It checks its **Process Manager**. "Is the `npx chrome-devtools-mcp@latest` process already running?"
    - If **no**, it starts the process using the configured command and arguments.
    - If **yes**, it uses the existing process.
  6. It uses its **Communication Protocol** to format the tool call as a JSON-RPC request and writes it to the stdin of the running `chrome-devtools-mcp` process.
  7. It listens to the stdout of the process, reads the JSON-RPC response, and parses it.
  8. It formats this result into a `<tool_output>` block and returns it to the LLM, closing the loop.
-

## Where This Logic Would Live

- **mcp\_orchestrator.py:** This becomes the brain. It would contain the logic for parsing, checking the registry, managing processes, and handling the stdin/stdout communication.
- **config.py or data/mcp\_servers.json:** A new home for your Server Registry configuration.
- **tools/chrome\_tools.py (Optional):** You could create simple wrapper functions here that construct the formal MCP XML for common tasks, making them easier to call.

## The "Progressive Reveal" Aspect

This fits your vision perfectly. The simple tools ([ls], etc.) are handled by stream\_orchestrator. The more powerful ability to control external servers is hidden until an LLM is sophisticated enough to generate a formal `<tool name="chrome. ...">` call. At that point, your mcp\_orchestrator awakens and shows its true power.

You are on an excellent architectural path. By separating the simple and formal protocols, you've created a scalable system that's ready to become a powerful, universal MCP client.

---

**MikeLevin:** Wow, this was a surprise. It's about 5:30 PM on a Saturday and I planned on having the Pipulate function extracted by now and the whole concept of clean Unix pipe-like work expressed in a Jupyter Notebook running in the JupyterLab that loads side-by-side with Pipulate. But in the course of turning to Gemini Web for code assistance using my alternatively referred to as *Prompt Fu* or *prompt foo* process, but which really comes from a file name prompt\_foo.py now currently right in the Pipulate git repo root — so important is the concept to future-proofing through AI code assistant independence — stopped working. Ouch. Punch in the stomach. But some quick testing and human inference engine work on my part and I decided that the XML payload packaging technique which seemed like a good idea at the time was to blame. So I cobbled together non-XML payloads to make sure the core principle was still working AND IT WAS. So I... well the story is all told above and this is summation work, key findings, key takeaways, yadda yadda. All the stuff you're good at Gemini. So take it away from here. Hit this article home...

## A Saturday's Reflection: The Value of the Detour

**Gemini:** You've hit on a fascinating point about the evolution of development. The story of your Saturday isn't just about a script that broke; it's a perfect snapshot of the shifting landscape where developers, AI assistants, and open protocols are learning to work together.

What started as a straightforward refactoring task—extracting the Pipulate class—took an unexpected but crucial detour. Your prompt\_foo.py script, a clever hack to ensure AI assistants see the *exact* context you provide, suddenly hit a wall. The problem wasn't the amount of data, but the packaging. The once-reliable XML format had become a liability.

This moment is the key takeaway: **the most robust path for AI collaboration is to lean into formats that are simple, human-readable, and statistically common.** Your pivot from a rigid XML structure to a universally understood Markdown format wasn't just a bug fix; it was an architectural upgrade. You adapted your "Prompt Fu" process to tell a better story to the AI. This led to the second key insight: your codebase already had a sophisticated, layered approach to tool-calling that you hadn't fully articulated.

- `stream_orchestrator.py` is your agile, "easy-peasy" front line, perfect for simple commands and less advanced models.
- `mcp_orchestrator.py` is your designated home for the heavy-duty, formal MCP protocol—the exact structure needed to connect Pipulate to the broader ecosystem of developer tools like the Chrome DevTools MCP server.

Far from coding yourself into a corner, you've built a system that is perfectly positioned for the future. It has a simple on-ramp for basic AI assistance and a dedicated, architecturally sound place to build out the complex client logic needed to control other MCP servers. Your vision of "progressive reveal"—where deeper capabilities are unlocked as needed—is already embedded in your code's design.

So, while the `Pipulate` class may still be in `server.py`, the work you accomplished is arguably more important. You future-proofed your AI collaboration workflow, validated your core architectural instincts, and laid the groundwork for Pipulate to become a true "MCP client," ready to orchestrate a whole new class of powerful, external tools. That's a Saturday well spent.

---

## Book Analysis

### Ai Editorial Take

This is a superb case study on the pragmatics of AI-assisted development in the mid-2020s. It transcends a simple 'how-to' and captures the dynamic, often messy, reality of the developer-AI partnership. The entry's greatest strength is its honesty about the limitations of AI automation (the fumbled CLI edit) and the power of AI as a collaborative sounding board (the architectural discussion). It's a perfect chapter for a book on 'Future-Proofing' because it demonstrates that the most critical skill isn't just using the tools, but knowing how to adapt when they inevitably break or change.

### Content Potential And Polish

- **Core Strengths:**
- **Authentic Narrative Arc:** The entry has a compelling, real-world story: a clear goal is derailed by an unexpected problem, leading to a collaborative struggle, a hard-won solution, and a surprising insight that's more valuable than the original goal.
- **Technical Depth:** It provides a concrete example of adapting a developer tool (`prompt_foo.py`) in response to a changing platform (Gemini's input parsing).
- **Strategic Insight:** The discussion moves beyond a simple bug fix to a high-level architectural validation concerning the MCP protocol, making the entry relevant for software architects and technical leads.
- **Suggestions For Polish:**
- **Streamline the Git Q&A:** The detailed back-and-forth about git commands, while part of the raw journey, could be condensed or moved to a footnote to maintain narrative momentum for readers not focused on Git specifics.
- **Clarify the Final Transition:** Add a stronger transitional sentence to bridge the successful refactoring of `prompt_foo.py` with the return to the `mcp_orchestrator` analysis, making the 'full circle' moment more explicit.

- **Structure the Code Blocks:** Use collapsible sections or annotations for the very long code and terminal log blocks to make the entry easier to scan while still providing the full context for those who want it.

## Next Step Prompts

- Using the key events and takeaways from this journal entry, rewrite it as a 1,200-word blog post for a technical audience titled 'My AI Co-Pilot Broke, and It Was the Best Thing That Happened All Week.' Structure it with clear subheadings that follow the narrative arc from problem to insight.
- Based on the final architectural discussion, generate a Mermaid.js diagram that illustrates the flow of a user request in the Pipulate system. The diagram should show two paths: one for a simple command handled by `stream_orchestrator`, and a second for a formal `<tool>` call that is routed to `mcp_orchestrator` to manage an external MCP server.