Building Agent projects without losing your mind

A clean, reusable structure that just works

MIGUEL OTERO PEDRIDO
JUN 18, 2025

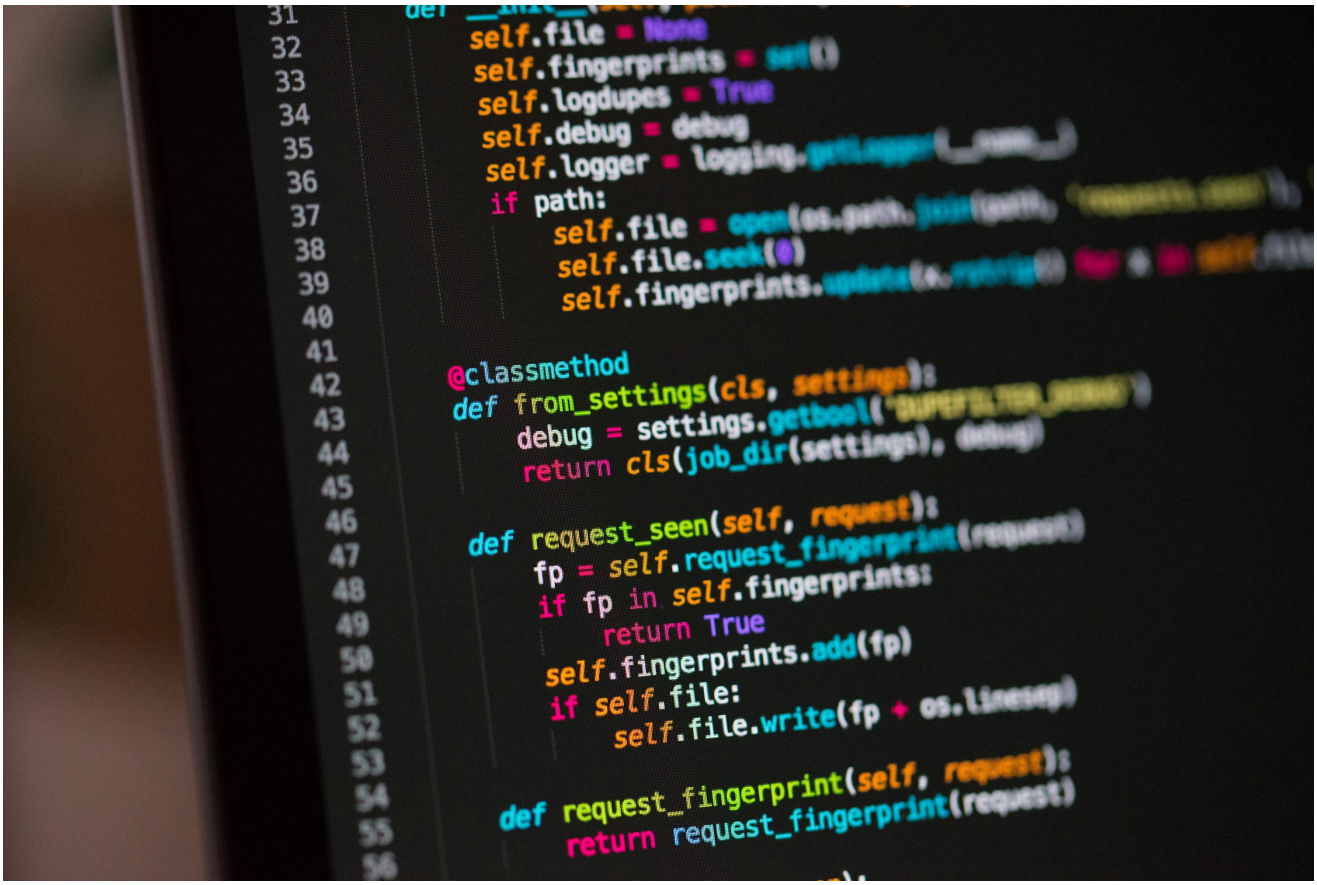


Image by Chris Ried (source: Unsplash)

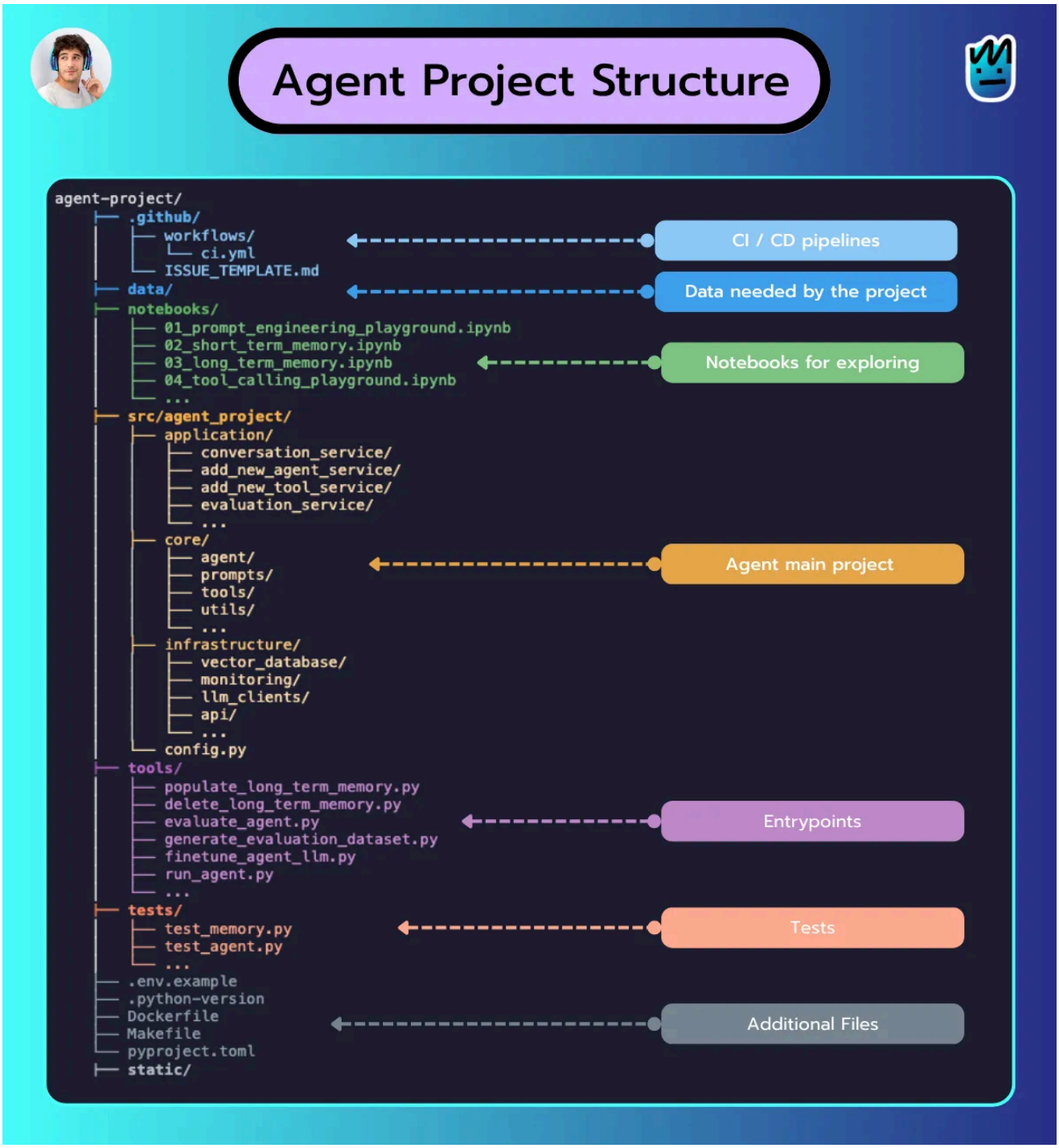
Ever since I started creating content, one question keeps coming up:

> How should I structure my Agent projects? 🤔

So in today's post, I'm sharing the structure I usually follow when starting a new of my end-to-end agent projects (e.g. PhiloAgents).

You definitely don't need to use *every* file or folder I mention, but this should give you a solid idea of how I like to organize things and think through software architecture.

The overall structure

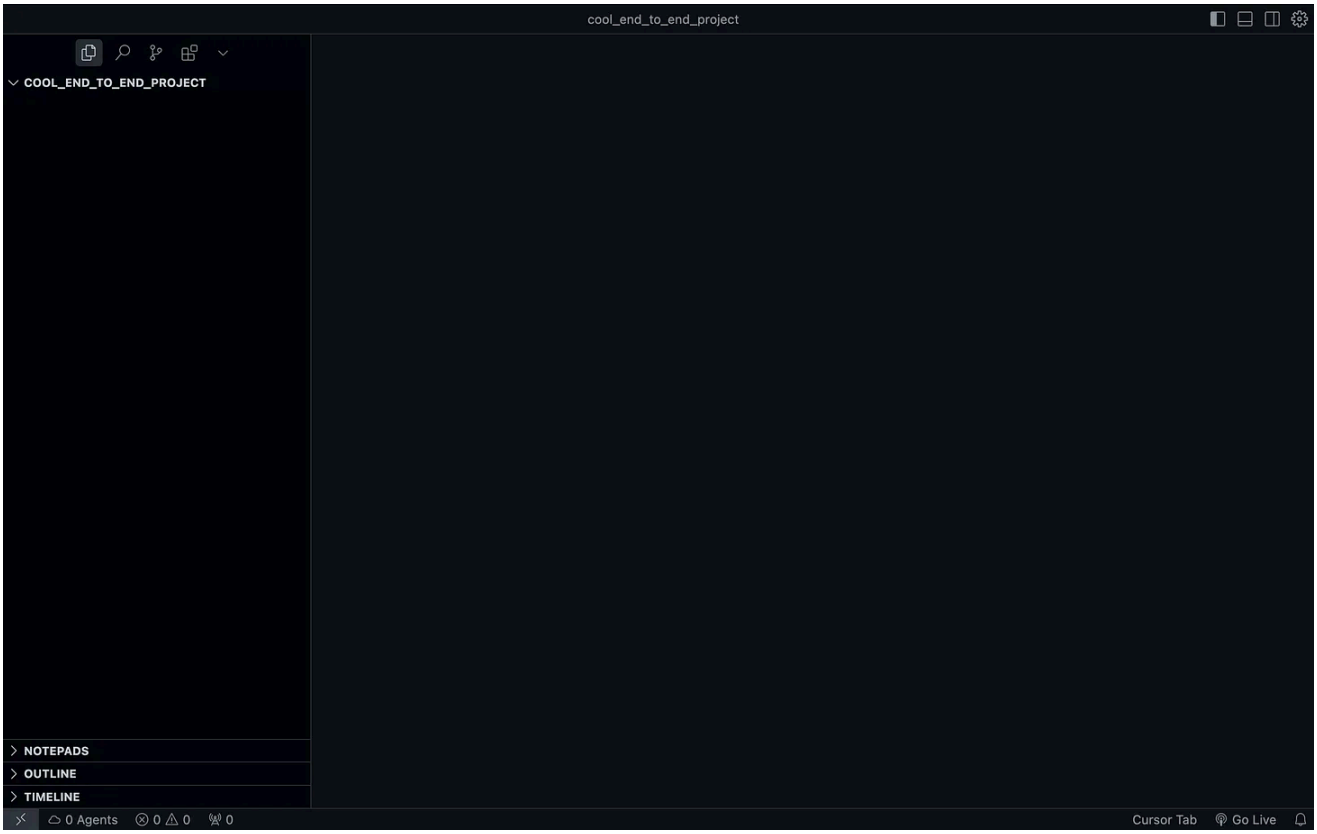


Agent Project Structure

In the structure above, you'll notice seven key components—ranging from CI/CD pipelines to core logic and testing.

For the rest of this article, I'll break down each one so you can see how they all fit together. Personally, the part I'm most interested in is the **Python library**— inside `src/agent_project`; that's where the heart of your application lives. It's where you define the logic that makes your agent actually *do* things.

Starting with a blank slate



The emptiness of space ...

Whenever I start a new Python project, it feels a bit like what Michelangelo said about sculpting: *"I saw the angel in the marble and carved until I set him free."* (sorry for the extremely nerd reference 🤣🤣🤣).

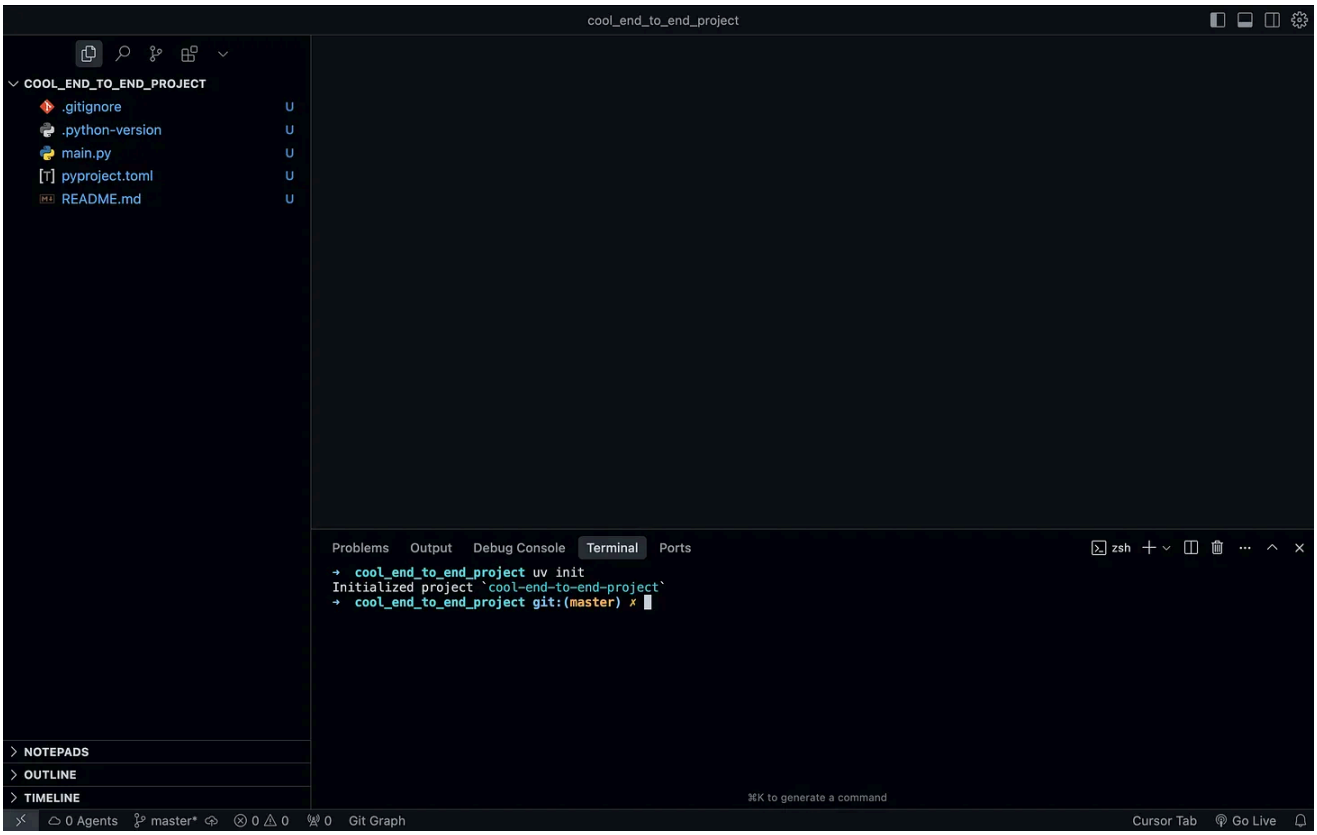
That blank project folder? It's my block of marble. The ideas are already in there—I just need to carve them out with code.

This is where the creative part begins. Not with a chisel, but with a terminal window and a keyboard.

And my first move?

It usually starts with **uv**. Just one simple command:

```
uv init
```



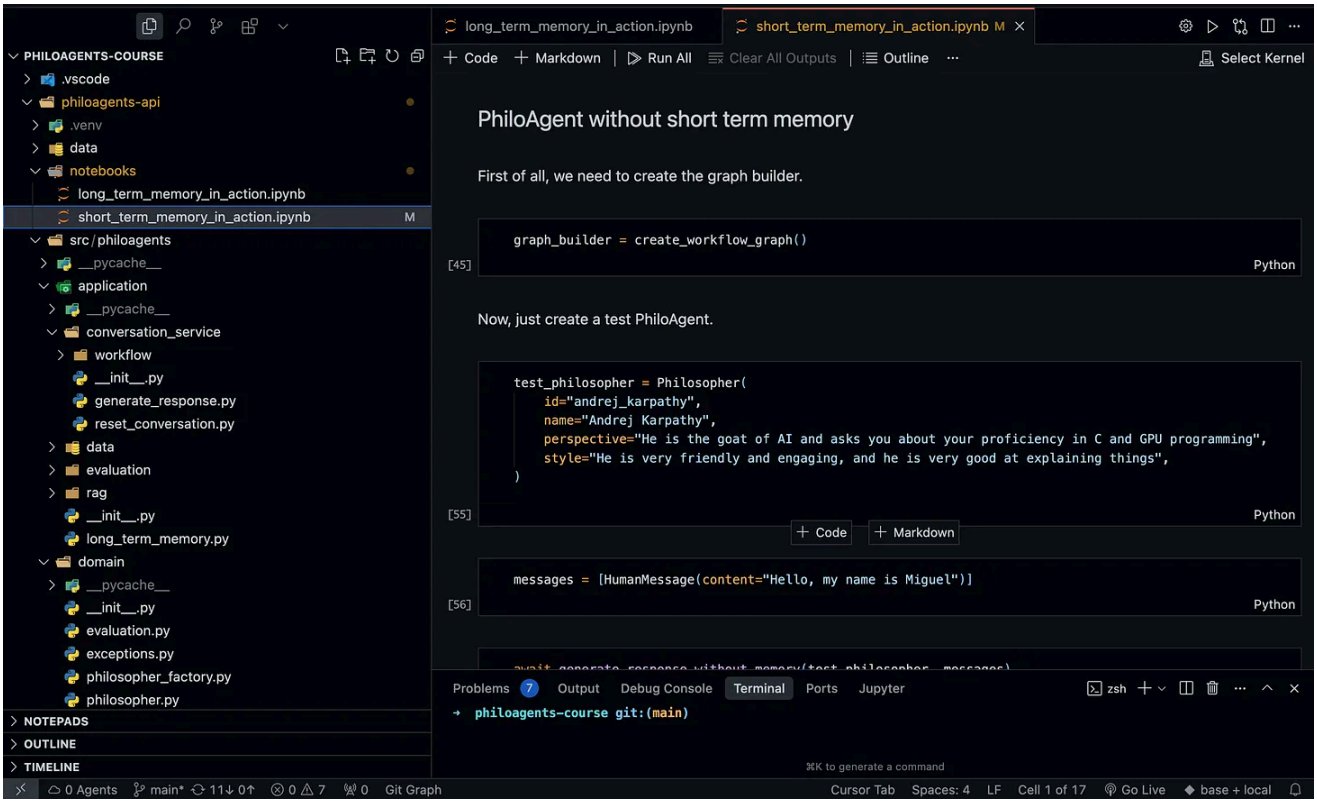
uv project structure

The files created at this stage line up with the “Additional files” section in the structure diagram.

For me, this is the *“let there be light”* moment in any Agent project. Once these core files are in place, things start to click. You’ve got a solid backbone to build on—without the usual dependency headaches that come with `requirements.txt` or `setup.py`. It’s clean, lightweight, and ready to grow with your code.

From here, it's all about layering in the logic, tools, and prompts that bring your agent to life.

Exploring before coding



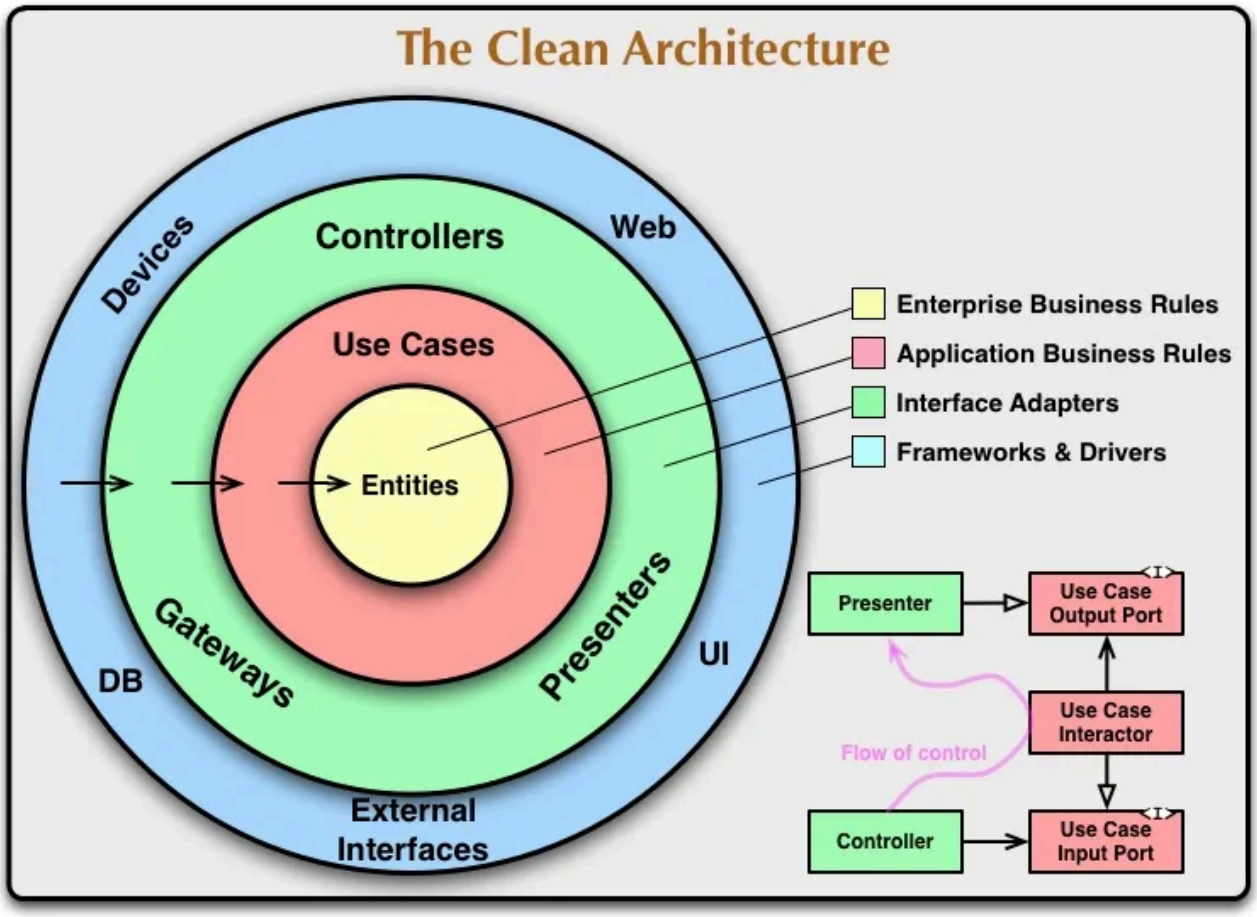
Exploring memory basic functionalities in notebooks (PhiloAgents project)

Maybe it's the Data Scientist in me, but before I dive into building the "real" system, I like to warm up with a bit of quick exploration.

That's why you'll always find a `notebooks/` folder in my repos. Whether I'm sketching out an agent workflow, testing a prompt, or poking at the behavior of a core class—I turn to notebooks in the early stages.

They're perfect for **fast feedback** without committing to full structure just yet.

The heart of the project - The Python library



For me, this is *the* most important part of the design—and honestly, since I started following this approach, I haven't looked back.

If you check the structure diagram, you'll see a `src/` directory that contains a folder like `agent_project/`. That's where all the core logic of the app lives.

If you've got a Software Engineering background, you might notice that the subfolder names aren't random—they follow a **Clean Architecture** approach. It's a way of organizing your code so it's easier to understand, test, and most importantly, change.

> Think of it like an onion: layers around a solid core.

Let's break down each layer:

🧠 Domain (Core) Layer

This is the heart of the system. In an Agent project, this is where I define all the key entities and objects. Some examples:

- Base classes for the agents *(note: no agentic frameworks here—just clean Python classes)*
- Prompts as plain strings
- Tools as raw Python functions (not LangChain or LlamIndex-style tools)
- Utility functions closely tied to the domain

⚙️ Application Layer

This is where we define what the system *does*. Think of it as the "use cases" layer. It's the services that drive your agents' behavior.

Examples:

- A conversation service, that handles interaction flow
- A RAG service, that pulls relevant data from vector databases
- An evaluation service, that scores or validates responses
- ...

🔌 Infrastructure Layer

Here's where the actual integrations live. All the tech your app *talks to*—databases, APIs, monitoring tools—goes here.

Examples:

- Connectors to Qdrant, Weaviate, MongoDB, etc.
- Clients for monitoring tools like Opik or LangFuse
- LLM clients (OpenAI, Claude, etc.)
- MCP server connectors and MCP client implementations
- ...

The beauty of this setup?

You can swap MongoDB for Qdrant or OpenAI for Claude, and everything still works—seamlessly.

Why?

Because your Domain and Application layers don't care about the specific infrastructure. They're built to be independent, and that flexibility makes scaling or changing tech a lot easier.

Your Agent needs an API. Period.

Chat and reset memory usecases (PhiloAgents)

Yes, technically the `api.py` file belongs to the infrastructure layer. But it deserves its own spotlight.

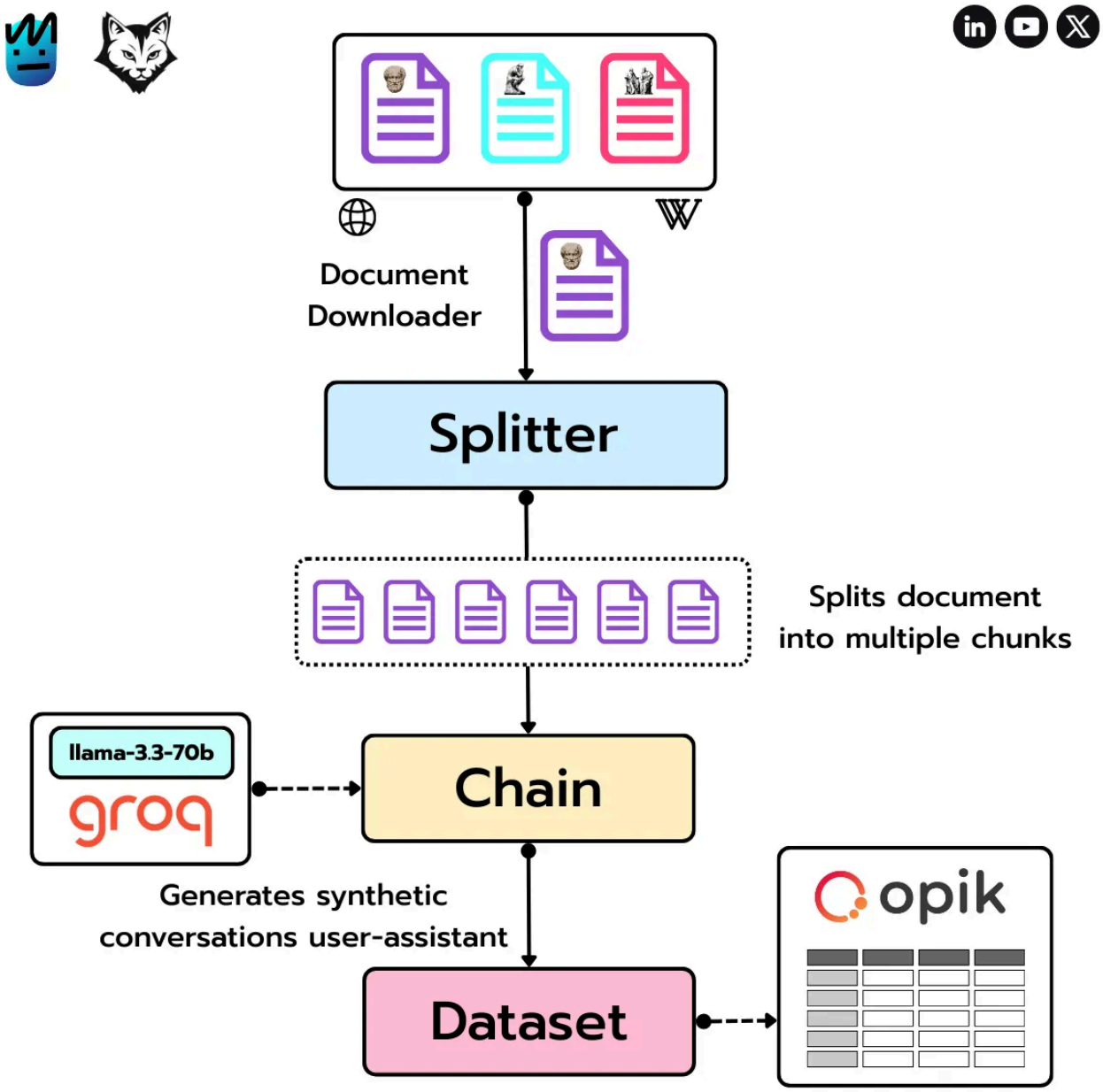
I keep seeing agent projects—solid, well-thought-out builds—that completely skip this one crucial step: **exposing the agent**.

If your agent isn't accessible, how are others supposed to use it? Whether it's another team, your frontend, or end users—*somebody* needs a way to interact with it.

That's why I always recommend thinking of your agent project as an API from the start.

And since we're following Clean Architecture, turning each application service into an endpoint becomes pretty straightforward (well, *sort of*). It's a natural fit—and it makes your project way more useful, scalable, and production-ready.

CLI Entrypoints: Don't skip these!



Besides the API, it's also a good idea to include a folder for **entrypoints**—in my case, it's named `tools/` in the root directory.

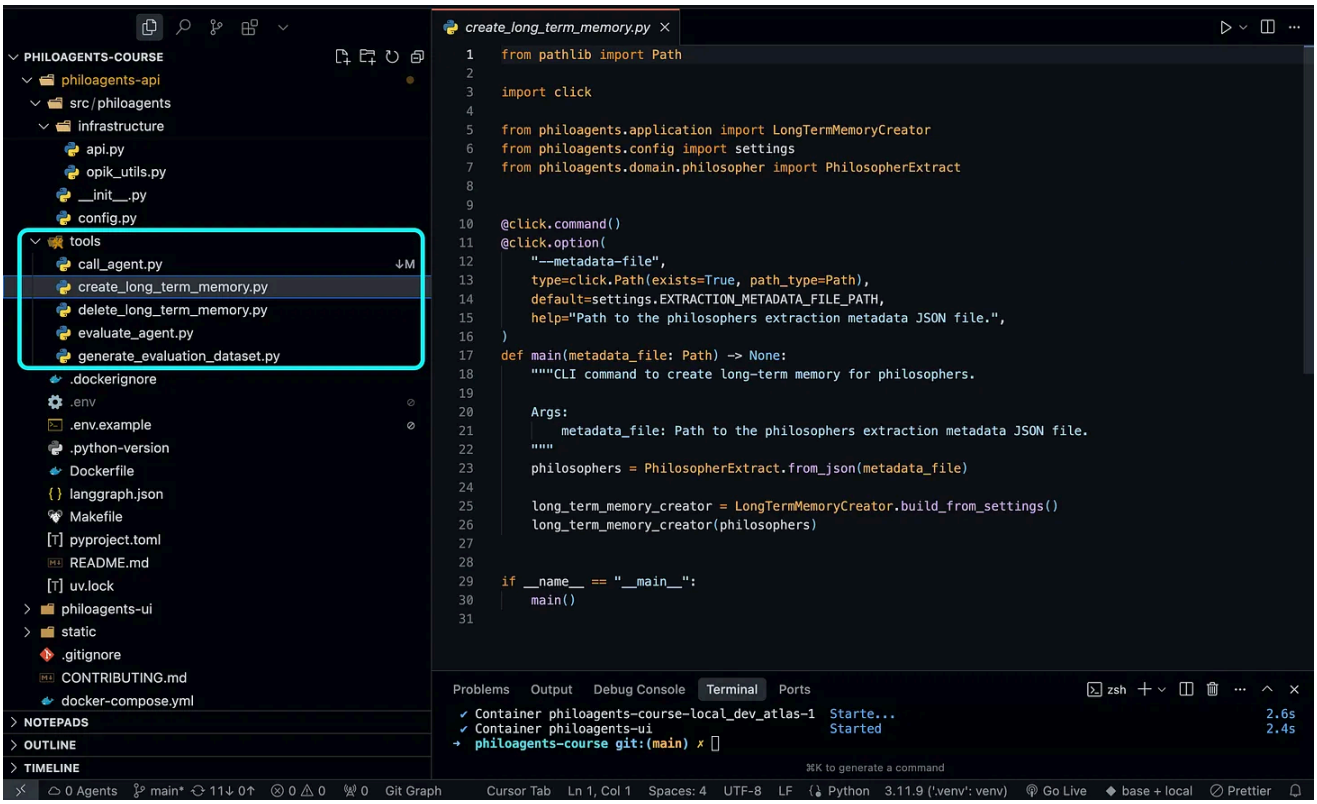
> Quick note: don't confuse this with *agent tools*. These aren't for your agent to use—these are scripts *you* run from the command line.

Think of them as CLI apps that tap into your Python project (which, thanks to `uv`, can be installed and used like any other Python package). These entrypoints are perfect for running common tasks or utilities tied to your agent system.

If you check out **PhiloAgents**, for example, you'll find tools that:

- Create the agent's long-term memory
- Trigger a sample agent run (great for testing if everything's wired up)
- Generate evaluation datasets

They're simple, practical, and help keep your workflow clean and reproducible.



Tests

Unit tests and integration tests are a must for any production-ready app—and agent projects are no exception.

I usually keep all my tests in a `tests/` folder, mirroring the structure of the main project. That way, it's crystal clear which part of the system each test is covering.

To make life easier, I also add `uv` commands in the `Makefile` so you can run tests manually with a single command. Super handy for quick checks before deploying or jumping to the next dev task.

CI / CD pipelines

I've worked with GitHub Actions, Jenkins, CircleCI—you name it. The goal is always the same: run all tests and validations before building a new version of the library (which ideally gets pushed to an Artifact Registry), and trigger the relevant MLOps or LLMOps pipelines.

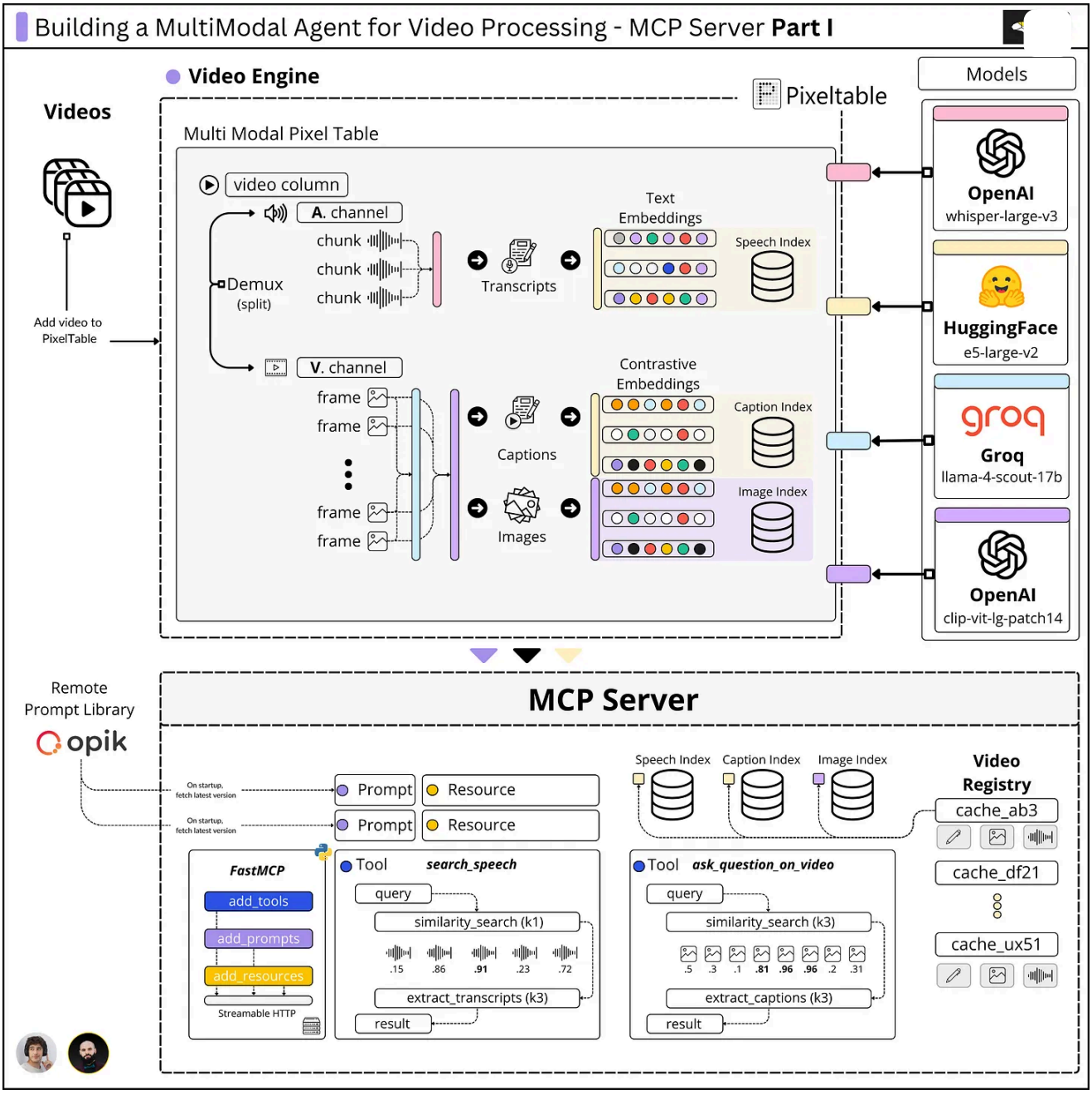
> We didn't fully cover this in **PhiloAgents**, but I've been thinking—would you be interested in a project that shows how to wire up these pipelines for things like testing, model versioning, LLM deployment, or MLOps best practices?

Phew, that was a long one!

But now you've got a full picture of how I approach agent project structure—from the first `uv` command to a clean, scalable architecture.

As always, I'd love to hear your thoughts. Got suggestions? Improvements? Drop them in. **You learn from me, I learn from you—it's a win-win** 😍

And before I go... a little teaser: a brand new **open-source course** is fresh out of the oven. Fully cooked, seasoned, and almost ready to serve.



This time: **MCP Servers** and **Video Processing Agents**.

And I won't be doing it solo—I'll be joined by the one and only Alex Razvant from Neural Bits .

Stay tuned 🍽️

Thanks for reading The Neural Maze! Subscribe for free to receive new posts and support my work.

63 Likes · 7 Restacks

← Previous

Discussion about this post

Comments Restacks

Write a comment...

Learning Journey Jun 18

♥ Liked by Miguel Otero Pedrido

I knew your next topic would be multimodal and MCP but didn't expect they are done in one course! 🤩 and YES for future courses on CI/CD pipelines and best practices🙌

LIKE (5) REPLY SHARE

1 reply by Miguel Otero Pedrido

Martyna Rachańczyk Martyna's Substack Jun 18

♥ Liked by Miguel Otero Pedrido

Please tell also about the structure of terraform or another IaC repos 🤩

LIKE (1) REPLY SHARE

4 replies by Miguel Otero Pedrido and others

6 more comments...