ChatGPT

Deep Research Task

Choosing Python and Preparing the Codex Environment (Rust and Lua Integration)

You have decided to use **Python** as the primary language for orchestration. This is a sensible choice given that your local primitives (ML frameworks, model serving libraries like **PyTorch** and **vLLM**) are Python-based. Python will allow easy integration with these libraries and with OpenAI's API. Meanwhile, **Codex** (your custom tool) is written in **Rust**, so you'll need to ensure the environment has a Rust toolchain installed for compiling and testing it. Setting up Rust (via `rustup` or similar) on the server or inside the container is important so you can recompile Codex whenever you make changes.

While **Lua** was considered, you plan to postpone its integration. The initial goal is to get a working Python API pipeline that can relay instructions to the OpenAI API and coordinate tasks. You might include a README or **instructions file** in the project directory to clarify the agents' objectives – essentially documenting the purpose of each component. This file can guide both human collaborators and any AI agents on how to use the system or where to leave notes if they are working on related sub-tasks. Maintaining such internal documentation will help orchestrate the multi-agent system and ensure everyone (and every agent) understands the project's goals.

Secure Remote Execution via SSH and Cloud Access

For remote execution, you will use a **direct SSH channel** to the cloud server (e.g. `ssh root@<server-ip>`). Since you're using the Codex web app as a controlling interface, it will need a secure token or key to authenticate with the remote server. This likely means using an SSH key pair or a short-lived access token. Once configured, the OpenAI-powered agent (through the Codex app) can programmatically SSH into your **DigitalOcean** bare-metal dev cloud instance. (This instance is an AMD GPU server, and you've arranged credit to keep it running for several months for this project.)

After establishing SSH access, the plan is to execute commands on the server in a non-interactive way via your Codex CLI. Specifically, you mention using a `codex exec` command – a non-interactive mode of the Codex CLI that can run shell commands in sequence. The idea is to have the OpenAI agent send **shell script instructions** directly to the server. For example, the agent might send a sequence of commands or a reference to a `.sh` script, and the Codex CLI would execute them on the server. All output would stream back to the agent, giving it feedback.

To manage these remote operations, you intend to implement **"watchers"**. A watcher is a process or agent that monitors the terminal output and system state during execution. It can read the stream of logs/console output up to a certain context limit. If the output becomes too large or a certain condition is met (like completion of a task or a time limit), the watcher will gracefully shut down the process or prompt the agent to stop. This prevents runaway processes and keeps the interaction within the token/context limits of the AI model. In summary, the SSH channel plus `codex exec` allows the AI agent to execute precise, short commands on the remote machine and immediately see results, while watchers ensure things don't get out of hand.

Incorporating OpenAI Embeddings and a Vector Store for Documentation

To give your AI agents access to the project's knowledge (like codebase documentation, large markdown files, etc.) without exceeding context limits, you plan to use **OpenAI's Embeddings** and a **vector store**. The workflow is as follows:

- **Cloning and Chunking Docs:** First, you would clone the relevant repository or collect all the large Markdown (`.md`) and text files that contain important information (for example, design docs, technical guides, or even code files if needed). These documents will be broken into smaller chunks (e.g. paragraphs or sections) so they can be embedded.
- **Creating Embeddings:** Using OpenAI's embedding API (for instance, the `text-embedding-ada-002` model), you convert each document chunk into a high-dimensional vector representation. This vector essentially captures the semantic meaning of the text in numerical form.
- **Storing in a Vector Database:** All these vectors are stored in a vector database or index (the **vector store**). This can be a managed service like Pinecone or Weaviate, but since you might want to avoid external dependencies initially, you can use an open-source solution or in-memory store. In fact, frameworks like *LlamaIndex* (GPT Index) offer a simple default vector store that can be saved locally to disk without any subscription [1]. This means you can keep your embeddings on your own server, and it's easy to scale up to a more robust database later if needed [1]. Another lightweight option is using **FAISS** (Facebook AI Similarity Search) for a local vector index, or even Redis in vector mode – there are guides to use Redis as a vector DB as well.
- **Semantic Search (Retrieval):** When the AI agent needs information (say, "What does the file 0_container.sh do?" or details from a README), it will formulate a query. This query is also embedded into a vector using the same model, and then the vector store is queried for similar vectors. The closest matches (i.e., the most relevant document chunks) are retrieved.
- **Providing Context to the Model:** Finally, those retrieved text chunks can be fed into the prompt for the AI agent. This way, the agent has the *relevant excerpt* from the large files, without having to load the entire content. The model will use this context to answer questions or make decisions. By using embeddings, the model can effectively "read" from the repository knowledge base on demand, working at a lower level than full natural language scanning – it's leveraging the semantic vector space to find information.

This approach ensures the AI agents can handle heavy documentation or code by searching through it, rather than exceeding token limits by always loading everything. It offloads part of the "memory" to an external store, which is a common technique for scaling AI assistants with private knowledge.

Containerized Orchestration with a Local vLLM Agent

You have a two-part setup for model orchestration: a local **agent** running on your server (likely utilizing vLLM for fast inference on the GPU), and the OpenAI API as another component. To manage the local agent environment, you will run a **Docker container** on the server. This container is configured for **ROCm** (so it can use AMD GPUs) and comes with vLLM and other dependencies pre-installed.

In practice, you will use a script (similar to `0_container.sh`) to manage the container lifecycle. According to the technical summary of that script, it stops any existing container and then launches a new ROCm-enabled Docker container with the project repository mounted inside [2]. In other words, each time you run this script, you ensure a fresh container is started (and the old one is terminated) so you have a clean, reproducible environment. The container uses a base image (for example, `rocm/`

`vllm-dev:nightly_0610_rc2_20250605`) that already contains vLLM and related libraries [3]. When it starts up, your repository (including scripts like the benchmarking tools, etc.) is mounted at a known location (e.g. `/workspace` inside the container) [3]. This means the container has access to all your project files and code.

Once the Docker container is running on the server, your OpenAI agent can **connect into it**. The method you described is using `docker exec -it codex-container /bin/bash` from the SSH session. Essentially, after the agent SSHes in, it can execute this command to drop into an interactive shell **inside** the running container. Now the agent has a direct line into the container's environment, where all the heavy ML libraries and the local LLM (via vLLM) are available. From this point on, the AI agent can issue Python commands, run the local model, execute scripts, or do any needed operations within the container.

This design cleanly separates concerns: the container encapsulates the local model serving and any experimental code (so you can install packages, run GPU workloads, etc., without polluting the host system), and the orchestration agent (via the OpenAI API) controls it externally through the SSH and Docker Exec bridge. It's a powerful approach because you can iterate on the container's contents (update the model or code) and simply restart it via `0_container.sh` to apply changes, while the orchestrator logic (in Python and Codex) remains on the outside managing the high-level process.

Additionally, the container approach simplifies scaling and monitoring. You could have logging on the container output, and your watcher processes can monitor the container's stdout/stderr. If something goes wrong inside (e.g., a process hangs or crashes), you can choose to tear down and restart the container without affecting the host. Since you're currently less concerned about multi-user security (it's just you using the system), running as root and attaching to the container is acceptable for now – though in a more secure setup you might create a limited user or use SSH keys with passphrases.

Inside the container, the **local orchestration agent** (perhaps a Python script or service you write) could be running continuously, or you might spawn it on demand. You mentioned an API to manually test functions from a client connection – this suggests you will expose some API endpoints (maybe via a FastAPI or Flask app, or even just through the CLI) that allows you to trigger certain actions for testing. Make sure to document these functions or endpoints in that agents' instruction file so you remember how to call them. By testing them manually, you ensure the building blocks are solid before letting the AI agent call them autonomously.

To summarize this step: you'll run a Docker container on the remote server to host your local LLM and tools. The OpenAI-powered agent connects via SSH and then attaches to this container's shell. From there, it can run any code or commands needed (including running the vLLM server, doing benchmarks, or orchestrating tasks). The `0_container.sh` script will be key to initialize this environment reliably each time (it stops any old instance and starts a fresh one with all dependencies loaded) [2] [3]. This gives you a controlled sandbox to execute code, which is particularly useful for running heavy GPU tasks or for isolating the experimental environment from your main system.

Guidance for Implementation and Next Steps (Support for a Non-Developer)

At this stage – with a complex setup spanning Rust, Python, Docker, remote GPUs, and OpenAI APIs – it's completely normal to feel a bit overwhelmed, especially if your background is in mathematics rather than software development. Here are a few suggestions and encouragements as you proceed:

- **Take it Step by Step:** Break the project into smaller milestones. For example, first ensure you can SSH into the server manually. Next, run the `0_container.sh` script and verify the container starts and you can execute basic commands in it. Then, test the OpenAI embedding by indexing a small sample document and querying it. By isolating each component (SSH, Docker container, embedding, local vLLM serving, etc.), you can troubleshoot issues in a focused way. This modular approach will build your confidence as each part starts working.
- **Leverage Community Resources:** A lot of what you're doing touches on cutting-edge dev ops and AI orchestration, but there are likely others who have done similar things. The repository you have (and its associated hackathon guide) is already a great resource – it shows how to set up vLLM in Docker and even how to benchmark it. Don't hesitate to refer to those guides (for example, the hackathon guide references how to connect VS Code via Remote SSH for easier development, which might be useful for you too [4]). Online communities (like forums or Reddit) can be helpful if you run into specific problems (e.g., Docker networking issues or embedding API quirks). For instance, if you eventually want to store vectors more permanently, you saw suggestions like using LlamaIndex or an open-source vector DB – those come from community knowledge that you can tap into.
- **Focus on Core Goals:** Remember the core motivation – *your prime number net*. If building this multi-agent, distributed setup is a means to an end (exploring a mathematical idea about prime numbers), keep that end goal in focus. It can be easy to get lost in the engineering complexity. If at times it feels like too much infrastructure, consider if there are shortcuts you can take. For example, if your prime-number project doesn't immediately need a high-performance local LLM, you might initially use the OpenAI API alone (which simplifies things) and only introduce the local vLLM when you truly need it for cost or speed reasons. There is a balance to strike; as the **"théorème de 3"** you quoted hints: *You always have a choice. You can exclude one option or another, or even both – but trying to exclude nothing (i.e. tackling everything at once) might end up excluding everyone (accomplishing nothing).* In practice, this means you might choose to postpone or drop certain features to focus on what matters most. It's okay to not do everything simultaneously.
- **Learning by Doing:** Since you identify primarily as a mathematician, think of this development process like learning a new branch of math – initially foreign, but gradually more logical. You'll likely pick up many new skills: server administration, containerization, API consumption, etc. Each of these will enrich your toolkit for future projects. Treat any stumbling blocks as part of the learning curve. With each hurdle overcome (be it figuring out an SSH key issue or getting a vector query to work), you're leveling up significantly.
- **Ask for Help When Needed:** Don't hesitate to use resources like Stack Overflow for error messages you can't resolve, or to ask colleagues/friends who are developers for a pointer. Even within your project, once the basics are running, your AI agents themselves might help with some coding tasks (for example, GPT-4 can often suggest how to fix a bug or improve a script if you provide the error and context).

Finally, as you proceed, remember to document what you build (in code comments or a journal). This will not only help any collaborators or AI agents working alongside you, but it will also help **you** recall why certain decisions were made. You're effectively orchestrating a symphony of different tools – **Python** for high-level logic, **Rust** for performance-critical or system-level tasks (Codex CLI), **Docker** for

isolation, **OpenAI services** for intelligence and embedding, and **vLLM** for local model inference. With patience and structured effort, these will come together to support your “bizarre prime number net” project.

Good luck with your development journey! Remember that you have the freedom to make choices on what to include or set aside – use that freedom wisely to avoid burnout. By focusing on one piece at a time, you’ll gradually build the entire system. In the end, seeing your mathematical idea come to life through this sophisticated setup will be very rewarding. Keep in mind the wise words of the *théorème de 3* and don’t try to juggle every ball at once. You’ve got this! [2] [3] [1]

---

[1] Vectorising documents and storing locally : r/OpenAI
https://www.reddit.com/r/OpenAI/comments/12yqkjs/vectorising_documents_and_storing_locally/

[2] AGENTS.md
https://github.com/Mikiish/cryptodex/blob/e28f1f626a84e98bbbf5c2e5487f153b2a98d95f/ai_hackmd_ressources/scripts/AGENTS.md

[3] [4] README.md
https://github.com/Mikiish/cryptodex/blob/e28f1f626a84e98bbbf5c2e5487f153b2a98d95f/ai_hackmd_ressources/hackathon_guides/1_developing_vllm/README.md