ChatGPT

Executive Summary

The **Vibe-code App** is envisioned as an AI-driven tool that auto-generates orchestration code (`CPAEngine` implementations) based on high-level functional intents. We explored multiple strategies to achieve reliable code generation that integrates seamlessly with VS Code, runs inside Docker, uses a Redis task queue, and fits into CI/CD workflows. Four leading approaches were examined: **(1)** leveraging OpenAI's Codex agent technology, **(2)** building a VS Code extension powered by GitHub Copilot, **(3)** adopting advanced AI coding editors like Cursor or Windsurf, and **(4)** using other AI code-generation toolchains (open-source agents or fine-tuned models). Each strategy's state-of-the-art capabilities, pros/cons, complexity, and feasibility for generating a container-ready `engine.py` have been analyzed. We recommend a **hybrid strategy** that combines the strengths of an autonomous code-generation agent (inspired by OpenAI Codex's iterative testing approach) with a VS Code extension interface for developer control. This offers the best balance of **reliability**, **integration**, and **maintainability**. We then present an implementation blueprint – from development setup and plugin development to agent orchestration logic and Docker/Redis integration – including example code snippets and CI pipeline steps. Finally, a phased **Next Steps** checklist is provided with estimated timelines.

Option Matrix

1. Fork/Adapt OpenAI's Codex Agent Project

Description: *OpenAI Codex* has evolved from a code-completion model into a full AI coding **agent** that can perform complex coding tasks autonomously. In 2025 OpenAI launched *Codex Cloud Agent*, a system where an AI can be assigned coding tasks and carry them out in an isolated environment [1] [2]. Unlike simple autocomplete, this agent reads the repository, writes new code, runs tests, and iterates on failures without constant human prompts [3]. OpenAI also open-sourced a **Codex CLI** tool (on GitHub) for autonomous coding in the terminal, which can be forked and customized. Adapting these for Vibe-code means using a powerful AI backend (e.g. GPT-4 or Codex-1 model) to generate the `CPA/engine.py` code given a high-level spec, possibly *iteratively refining the code until it works*.

Pros:

- *State-of-the-art capability:* The Codex agent is at the cutting edge of AI code generation. It can **independently write and verify code** rather than just suggest completions [3]. Notably, it *can run tests and adjust code until tests pass*, which is ideal for ensuring a generated `engine.py` actually executes correctly in a Docker/Redis setup [3]. This yields high reliability in the produced logic.

- *Multi-step autonomy:* Codex can handle complex tasks in stages. For example, it can write code, run the code inside a sandbox container, observe errors or failing tests, and then fix its own mistakes in subsequent iterations [4] [3]. This autonomous feedback loop reduces human debugging effort.

- *Open-source starting point:* OpenAI's **Codex CLI** (open-sourced on GitHub) provides a base to build on. It's essentially a "lightweight coding agent" that supports multiple AI providers (OpenAI, Anthropic, local models, etc.) [5] [6]. We could fork this project to create our own domain-specific code generator. Being open source (Apache-2.0 licensed [7]), it's highly customizable – e.g., we can embed our `IEngine` interface definitions or templates into its prompting logic.

- *Provider flexibility:* The Codex CLI and similar tools allow using different models or API endpoints [5]. We could start with OpenAI's best (GPT-4-based) models for quality, and later switch to self-hosted models with minimal change. This avoids lock-in and can manage cost by choosing model per use-case.

- *High-quality code output:* Codex (the model) was trained on massive code corpora and optimized for following instructions precisely [3]. It tends to produce code that matches human style and project conventions. For example, it was reinforced to mirror typical pull request coding styles [3] – helpful for maintainability of generated engines.

Cons:

- *Access and cost:* OpenAI's new Codex agent is a **proprietary cloud service** at present. Gaining direct access might require an enterprise arrangement or API that isn't publicly self-hostable. This could incur significant **usage costs** (Codex likely relies on GPT-4 or similar, which are expensive per token). Each code generation (with iterative test runs) could be a lengthy session, consuming many tokens and thus cloud credits. If used heavily, costs may pile up, and budgeting for API usage or Codex subscriptions is needed. There's also potential vendor lock-in if our solution heavily relies on this closed platform.

- *Integration complexity:* While forking the open-source Codex CLI is possible, it's a **complex project** (nearly 30k stars, multi-language codebase [8] [9]). Adapting it to our needs may require significant understanding of its architecture (Rust/TypeScript core). Tuning its "agent" to always produce a correct `CPAEngine` structure (with our specific methods like `run(), plan(), analysis(), etc.) will require careful prompt engineering or code changes in the agent's planning logic. This is doable but non-trivial.

- *Hardware/infra requirements:* The Codex agent runs tasks in isolated containers in the cloud [4]. If we fork it for self-hosting, we'll need infrastructure to run similar sandboxed environments (likely Docker containers or VM instances to execute code safely). This adds complexity to our setup. We also must ensure our Docker environment (for running the generated engine and its tests) is configured similarly to how Codex expects (e.g., accessible codebase, test scripts).

- *Latency:* Autonomous iteration can be time-consuming. Codex tasks can take *1–30 minutes* for complex prompts [4]. Generating an orchestration engine might lean toward the longer side if multiple agent interactions or test loops are needed. This means developers might wait minutes for code generation to complete, which could hurt the interactive experience (though it's acceptable if automated offline or run in background).

- *Overkill for simple changes:* Using a full autonomous agent might be overkill if the functional intention is minor. The agent might end up exploring or modifying parts of the codebase unnecessarily if not

perfectly constrained, potentially introducing errors in unrelated code. Ensuring it stays *focused only on* `CPA/engine.py` and relevant files is important (which we could handle by scoping its access to just that folder).

Cost & Complexity: Using OpenAI's Codex service would likely come under a paid plan (possibly enterprise pricing). Each generation run would incur OpenAI API costs. The exact pricing of Codex-as-agent isn't public, but since it runs GPT-4-level models with long context and tool usage, it's expected to be **quite expensive per run** (perhaps on the order of several cents or dollars per generation, depending on code size and iterations). Forking the open-source Codex CLI avoids licensing cost, but integrating it is a **complex engineering project** requiring expertise in AI agent frameworks, and possibly modification of its default behaviors. We would need to invest engineering time to configure prompts, tools (e.g., giving it a command to run our test suite), and to maintain this fork as OpenAI updates it.

Feasibility for `CPAEngine` **generation:** Technically high. Codex's ability to read multiple files and follow instructions means we can feed it the `IEngine` interface and an example implementation as context. It should be able to produce a new `CPAEngine` that *conforms to our template* and orchestrates calls to AI agents as instructed. Thanks to iterative test-running, it can verify that the new engine runs (for example, it could execute a sample Redis task through the engine inside a container and check the outcome). This dramatically increases confidence that the generated `engine.py` will be container-ready and functional. The main feasibility concern is access and setup – if we can navigate the cost and integration hurdles, this approach offers a very powerful solution [4] [3].

2. VS Code Extension + GitHub Copilot

Description: This strategy centers on a **custom VS Code extension** that leverages GitHub Copilot (or the underlying OpenAI Codex model via API) to assist or automate writing the `engine.py`. In practice, we'd build a VS Code plugin (likely in TypeScript) that provides commands to "Generate CPAEngine" logic. When triggered, it could gather input from the user about the desired functionality (e.g. a brief description of the workflow or agents to orchestrate) and then use Copilot to produce the code. GitHub Copilot is an AI pair-programmer integrated in VS Code that provides code suggestions by analyzing the context in the editor [10]. We can exploit its strengths by feeding relevant context: for instance, the extension can open the `IEngine` interface definition and a template class, insert a commented outline of steps (analysis, plan, run, etc.), then prompt Copilot to fill in the implementation. Alternatively, the extension could utilize the Copilot Chat API (if available) or OpenAI's API directly to generate the code snippet and insert it into the editor.

Pros:

- *Seamless developer experience:* VS Code is where developers already spend their time. A **dedicated extension** means generating an `engine.py` becomes an integrated action – e.g. a command palette option like "> Vibe-code: Generate Engine". The code appears right in the editor for immediate review and editing. This shortens the loop from generation to testing. Copilot is already "in the flow" of coding, making the AI assistance feel natural.

- *Leverages Copilot's training:* GitHub Copilot is powered by OpenAI's Codex models and has been trained on a vast corpus of code. It excels at writing code that fits with existing context and comments [11]. By providing the right cues (e.g. docstring "# Orchestrate two agents: agent1 for

planning, agent2 for execution, using Redis queue…"), we can get high-quality boilerplate code quickly. Copilot suggestions often follow common patterns and might have seen similar orchestrator code, so it could generate a substantial portion of `engine.py` correctly with minimal prompt.

- *Interactive control:* Unlike a fully autonomous agent, the **developer remains in control** with Copilot. The extension can guide the generation step by step. For example, it might first generate a skeleton class with method stubs, then ask Copilot (or prompt the user to ask Copilot Chat) to implement each section. The dev can inspect and tweak in between. This human-in-the-loop approach can yield reliable results with less risk of the AI wandering off-spec. It's also easier to correct small mistakes on the fly – the developer can simply edit or prompt Copilot again for a fix.

- *Lower complexity to implement:* Developing a VS Code extension that calls an API or utilizes existing Copilot functionality is a **moderate complexity** task, well-documented in VS Code's docs. We don't need to build the AI logic from scratch – just glue existing services into the IDE. For instance, we can call the OpenAI API from the extension (with an API key) relatively easily, as many open-source VS Code AI extensions do [12]. This means faster development compared to creating a whole new AI agent framework.

- *Cost-effective for user:* If the user already has a Copilot subscription, using it in VS Code has **fixed predictable cost** (Copilot Pro ~$10/month). This is cheaper and easier than paying per API call. Even using the OpenAI API via the extension could be cost-controlled by only invoking when needed (and perhaps using cheaper model variants for drafts). Also, no extra cloud infrastructure is needed – VS Code and Copilot handle the heavy lifting, with GitHub's cloud doing the model inference.

- *Context awareness:* Copilot automatically considers the open project files and context when generating suggestions [11]. This means the extension approach can naturally include relevant project context (the domain model classes, etc.) by having them open or by programmatically providing them to Copilot's context window. The result is code that better fits the existing codebase (naming, style, framework usage).

Cons:

- *Limited autonomy:* Copilot (on its own) will not iterate and test the code. It provides *one-shot suggestions*. If the initial suggestion has bugs or omissions, it's up to the developer to detect and prompt again. The extension could be enhanced to run the code or tests after generation, but handling multi-step fixes would require custom logic (or the developer manually invoking Copilot Chat to debug). In short, it's not as hands-free as an agent: **human oversight is needed** to ensure the generated `engine.py` is correct and robust.

- *Copilot API limitations:* GitHub Copilot does not expose a public API to third-party extensions. We rely on either hacky workarounds or user action (e.g. the extension might insert a comment and rely on Copilot's normal suggestion mechanism). This could be brittle. Alternatively, using OpenAI's API directly from the extension is possible, but then we forego Copilot's specific tuning and also must handle things like rate limiting, context limits, etc. We would also need to manage the prompt construction carefully. This adds development effort and complexity (essentially writing our own mini-Copilot logic).

- *Quality variance:* While Copilot is often impressive, it **may produce imperfect code** especially for novel scenarios. Orchestration of multiple AI agents via Redis is a specific task it might not have seen frequently. It could make incorrect assumptions or omit important parts (like error handling, or the correct usage of our domain classes). Ensuring it produces *container-ready and secure code* might require providing significant guidance in the prompt. There's a risk of "hallucinated" code that looks plausible but doesn't actually work with our system – careful review is needed.

- *Security/compliance:* Using Copilot means sending our code context to a third-party (GitHub/OpenAI) cloud. If our project is proprietary, this might be a concern (though GitHub offers Copilot for Business with policy controls). It also means relying on an external service which may occasionally be down or slow. That said, Copilot has enterprise options to mitigate data leakage (e.g. no training on your prompts).

- *No built-in test execution:* Copilot won't run the code it writes. We will have to separately run the app to see if the engine works. So the burden of validation shifts to our testing pipeline or the developer. In contrast, other approaches (Codex agent or GPT-engineer) could potentially run a test automatically. Here, we'd incorporate that either via the extension prompting the user to run tests, or leaving it to CI.

Cost & Complexity: The **development complexity** for this option is moderate. Writing a VS Code extension that calls an API (OpenAI) or orchestrates Copilot is straightforward if we keep it simple: a command that collects input and shows output. If we attempt deeper integration (like listening to file events or automating Copilot's ghost text), complexity increases. We also must handle authentication (Copilot uses the user's login, whereas OpenAI API would need the user's API key or our service key). On the **cost** side, Copilot for individuals is ~$10/month (free for some students/OSS) and for business $19/user/month – relatively low [13]. The OpenAI API alternative would charge per usage (GPT-4 is ~$0.06-0.12 per 1K tokens as of 2025), but generating an engine might only consume perhaps a few thousand tokens, i.e. pennies per use – negligible for infrequent runs. Overall, this approach is cost-effective and predictable.

Feasibility for `CPAEngine`: Feasible, with some caveats. Copilot can definitely generate a class that implements a given interface; for example, if we prompt "Implement a CPAEngine that receives tasks from Redis and uses two AI agents to process data," we can get a starting point. Copilot's strength in using context means if the `IEngine` base class and a sample engine are in the project, it will try to follow that pattern [11]. The extension can ensure those are visible. The result will likely require a round of tweaking and testing, but should mostly adhere to the required template (methods like `run(), plan(), etc.`, present because the class signature and parent class demand them). Ensuring the code is *container-ready* (e.g. proper Redis connection handling, no hardcoded paths) might require us to include such details in the prompt or as a snippet for Copilot to follow. In summary, this approach can **quickly draft the engine code** and integrate with the developer's workflow, but will rely on the developer (and subsequent pipeline tests) to catch any issues. It's less autonomous but **very user-friendly** and relatively low-risk to try.

3. Cursor or Windsurf AI Code Editor Approach

Description: *Cursor* and *Windsurf (formerly Codeium)* are advanced AI-powered IDEs – essentially VS Code forks enhanced with AI features and agentic workflows. They provide an even more powerful environment for code generation than vanilla VS Code+Copilot. Notably, both support **multi-file AI edits, chat with the codebase, and one-click "fix" or "generate" actions** throughout the IDE [14] [15]. Each has an "agent

mode": Cursor's Composer agent and Windsurf's Cascade. These allow the AI to operate more autonomously on code, e.g. generating code across multiple files, running commands (like tests or shell commands) and using broader context without the user explicitly providing it [16]. The strategy here would be to *either* use these tools directly to create our `engine.py` (i.e. adopt them in our dev process) *or* replicate their techniques in our own tool. For instance, a developer could open the c0engines repository in Cursor, write a high-level spec in natural language, and let the agent mode produce a new `CPA/engine.py`, possibly even executing it to verify outputs. Another angle is to study how these editors implement agentic coding (they often integrate something like an open-source agent extension) and incorporate similar logic into our Vibe-code App.

Pros:

- *Rich agentic capabilities:* Cursor and Windsurf blur the line between human and AI coding. With **agent modes**, they can automatically handle context and even execute code. For example, *Cursor's agent* can "**generate code across multiple files, run commands, and figure out what context it needs**" without the user manually selecting files [16]. Similarly, Windsurf's Cascade can orchestrate steps. This means for our use-case, these tools could not only write `engine.py` but also modify or create related files (e.g. Dockerfile or config) if needed, and run shell commands to test it – all within one continuous AI-driven flow. This yields a thorough, working implementation with minimal manual steps.

- *IDE integration with testing:* Both editors allow the AI to use an integrated **terminal/console**. There's an extension called *Cline* (which works in these IDEs and VS Code) that serves as an autonomous coding agent, able to execute commands to validate its output [17]. Using such an extension or the built-in agent, the AI can, for instance, run `pytest` or a sample script inside the IDE to verify that `engine.py` processes a dummy task correctly. This is akin to having an AI that writes code *and immediately checks it* by running in a sandbox, which greatly improves reliability of the generated logic. It's very much in spirit with our needs for container-ready correctness.

- *Multi-file context and editing:* If the orchestrator logic touches multiple pieces (say `engine.py` plus some supporting module or a manifest), these AI IDEs handle it gracefully. They maintain a shared context of the entire project and can open/write multiple files in one AI instruction. For instance, one could prompt "Create a new CPAEngine class implementing IEngine to orchestrate Agents X and Y, and ensure any new dependencies are added to requirements.txt." The agent could then produce the class in `engine.py` and also modify `requirements.txt` in the same go. This **holistic project view** is a big advantage over single-file generation approaches.

- *Enhanced UI/UX for AI coding:* Windsurf and Cursor come with many quality-of-life features for AI-assisted development – e.g. **inline suggestions, chat explanations, "fix with AI" buttons on errors, etc.** [14]. This makes the process of refining the generated code much smoother. If the first pass isn't perfect, one can highlight a problematic block and ask the AI to improve it. The tools often maintain a history of AI interactions, so you can backtrack or review why the AI made certain changes, which aids debugging. Essentially, they provide a robust environment to **iteratively converge on a correct solution** with AI help.

- *Potentially lower cost (for Windsurf):* Windsurf (Codeium) historically offered a **free tier** for individuals, with on-device or cloud-based models. As of late 2024, Windsurf's pricing involves "model flow action

credits” [18] – many basic completions are free/unlimited, and only heavy usage consumes credits (with generous quotas). This could mean we get a lot of AI assistance without direct cost, which is attractive for experimentation. Cursor is paid ($20/user/month) [18] but that’s still a fixed reasonable cost for extensive capabilities. Both allow using custom/self-hosted model endpoints as well, which could reduce operational cost if we bring our own AI.

Cons:

- *Adopting a new IDE:* The team would need to adopt either Cursor or Windsurf as their coding environment for this purpose. This might not be a big hurdle (since they are VS Code forks, the learning curve is small), but some developers may resist switching editors. Additionally, if our development workflow is tightly integrated with standard VS Code (extensions, settings, etc.), migrating those to a forked IDE could pose minor inconveniences. It’s an extra tool to manage, update, and trust.

- *Not a standalone “app”:* If we rely on these tools, the **Vibe-code App might simply become a workflow recommendation** (“use Cursor to generate code”) rather than a distinct product we build. The question implies building our own application, so using Cursor/Windsurf directly might be tangential unless we integrate with them. We could write our own extension within those IDEs (they support VS Code extensions [19]), but then we could also just do that in VS Code itself. In short, this strategy might dilute the identity of our solution, unless framed as “Vibe-code powered by Windsurf” or similar.

- *Complexity to replicate: If we interpret “replicate Cursor/Windsurf flows” as building similar capabilities ourselves, that is a massive undertaking**. These editors are the result of large efforts (Cursor is loaded with “power features” [20]). Recreating their multi-file agent, context management, and polished UI from scratch is likely infeasible in the short term. We would be better off leveraging existing open components (like the Cline agent extension) rather than reinventing the wheel.

- *Uncertain maturity of agent mode:* While promising, the “agent” features in these IDEs are fairly new. Reviews indicate that they sometimes misfire or require user guidance to proceed (e.g., Cursor’s agent might need the user to approve file changes, which interrupts full automation [21]). The builder.io comparison noted that it wasn’t clear if Cursor/Windsurf’s so-called agents are “truly agents” in the autonomous sense [19] – they may still lean on the user to confirm actions. So, the dream of one-click fully correct code might not always match reality; some manual iteration could still be needed.

- *Resource usage:* Running these AI-augmented IDEs can be heavy on local resources if using local models, or they may require constant cloud calls for the AI. In practice this is similar to Copilot usage, but agentic operations that run tests will use CPU/Memory on your machine or a cloud container. Developers need a reasonably powerful dev machine or cloud setup to use them effectively, especially if the project grows large (context management for big codebases can strain the model and the IDE).

Cost & Complexity: If simply **using** Cursor or Windsurf, the cost is low: Cursor at ~$20/month/user [18] and Windsurf possibly free or credit-based for individuals. That might actually be cheaper than heavy OpenAI API usage for equivalent work. Building on them (via extensions) is also easier than starting fresh, since we

can assume their AI capabilities exist and just interface with them. However, if we go for a **self-hosted approach** inspired by them, complexity spikes. We might try combining open components: for example, use **VS Code + Cline extension** (Cline is an open-source autonomous coding agent with Plan/Act modes [22]) to mimic what Cursor does. This could be a middle-ground: it's complex but feasible to set up Cline and configure it for our needs, and we avoid depending on proprietary IDEs. Still, configuring such agent extensions demands deep understanding of prompting and possibly writing custom "skills" or instructions for our domain. In summary, leveraging these tools directly is low-complexity, while reimplementing their functionality ranges from moderate (using existing open agents) to high (writing our own from scratch).

Feasibility for `CPAEngine`: Very high, if using the tools directly. One could realistically open our project in Cursor, open a chat and say: *"Create a new engine that implements IEngine. It should take tasks from a Redis queue, use AgentA to analyze the task, then AgentB to execute a plan, and return the result. Make sure to integrate the result into the domain.Request object."* Based on Cursor's capabilities, it can gather the context (it might automatically include `IEngine` interface and relevant domain classes in the prompt) and draft the code. If its agent mode is activated, it might even run the code or at least a linter, then inform us of any issues or automatically fix them. This covers both code generation and a degree of validation. The **output quality** is likely to be good since these tools often utilize GPT-4 or Claude under the hood (Windsurf, for example, can use Anthropic's Claude models which are strong in coding [23]). The main feasibility consideration is whether we want to build our solution around an external editor. As a short-term boost or proof-of-concept, these are great: we can quickly prototype `engine.py` using them. For the long term, we'd extract the lessons (multi-step generation, integrate testing) into our own pipeline. In conclusion, the Cursor/Windsurf approach can definitely produce a working `CPAEngine` and may inform our ultimate implementation, but it might serve more as a *power tool in a developer's hands* than a standalone "app" we deliver.

4. Other AI-Based Toolchains and Approaches

Description: Beyond the above, there are other plausible ways to implement an AI-driven code orchestrator. These include using open-source **"AI developer"** tools, fine-tuning our own models, or hybrid pipeline approaches. One notable path is adopting projects like **GPT-Engineer** or **aider**. *GPT-Engineer* is an open-source CLI tool that takes a natural language prompt and generates an entire codebase (or feature) through a series of reasoning steps [24]. It can even execute the code and ask for improvements, similar to the Codex agent. Another idea is using a **local code model** (such as Meta's *Code Llama* or BigCode's *StarCoder*) tailored to our needs. We could fine-tune such a model on examples of `engine.py` or similar orchestration code so it learns the pattern, then host it in a container to generate new variants on-demand. Additionally, we can design a custom pipeline with frameworks like *LangChain* or *Semantic Kernel*, where the AI goes through planned stages: (1) read spec, (2) generate code, (3) run code in Docker, (4) analyze results, (5) refine code if needed. This "agent loop" can be implemented with existing Python libraries and a reliable LLM backend.

Pros:

- *Open-source control:* Using tools like GPT-Engineer or a fine-tuned local model means **we control the code and model**. GPT-Engineer is MIT-licensed and hackable – it even suggests using aider for a maintained CLI [25]. We can modify its workflow scripts to suit our exact use case. This avoids dependency on any proprietary service and ensures longevity of our solution. Similarly, having our own model (or a community model) means we aren't rate-limited or constrained by external policies.

- *Purpose-built workflow:* GPT-Engineer's philosophy is directly aligned with our needs: "Specify software in natural language, and *watch as an AI writes and executes the code*, then ask for improvements" [24]. It is designed to create a working project from a prompt, even running tests in the process. This means much of the logic for an agentic workflow is already implemented. We can tweak its "steps" (it typically generates a plan, then code, then runs it) to specifically generate `engine.py` and then invoke a test (perhaps running the engine in Docker with a dummy task). This yields a **container-ready orchestration logic** with verification in one pipeline run.

- *Multi-agent or tool use:* We can incorporate the concept of *tools* for the AI. For example, using LangChain, we could give the LLM the ability to call a Python function that spins up a Docker container with Redis and our generated engine to test it. The AI can observe the output (success or error) and then adjust the code accordingly. This is essentially building a specialized AutoGPT-like agent for our problem domain. It's complex, but frameworks exist to ease this (LangChain's toolkit, OpenAI's function-calling APIs, etc., can be harnessed).

- *Customizability and maintainability:* Because these approaches involve writing or tweaking code ourselves, they are highly **maintainable** in the sense that our team can fix bugs or adjust logic without waiting on a vendor. Fine-tuning a model on our code conventions could also improve maintainability of outputs (the model will produce code closer to our style and requirements). Over time, we can incorporate feedback and new examples into the model or prompting strategy, steadily improving the reliability of generation.

- *Performance improvements with open models:* Open models are rapidly improving. Meta's **Code Llama** (34B and 70B variants) has achieved near state-of-the-art performance among code models, scoring ~53–67% on HumanEval benchmarks (close to OpenAI Codex's level) [26]. These models are available under permissive licenses and can be run on powerful hardware or via optimized server setups. Using such a model could drastically cut per-use costs (just infrastructure cost) and eliminate external API latency. It also sidesteps data privacy concerns since everything runs in-house. If our usage is high, investing in a custom model might pay off in the long run.

Cons:

- *Engineering effort and complexity:* While open solutions avoid license costs, they impose a larger **initial engineering burden**. Setting up GPT-Engineer or a similar agent requires understanding its configuration and possibly writing new "strategies" or prompt templates for our specific task. Fine-tuning a model requires collecting training data (we'd need examples of orchestrator code – which we might have only a few of, unless we generate synthetic ones). The ML expertise and compute required for fine-tuning a 34B parameter model are non-trivial. If we go with a custom LangChain agent, we'll be writing and debugging prompt logic, which can become complex as the agent scales (ensuring it doesn't get stuck or hallucinate tools usage, etc.).

- *Reliability and test depth:* Not all open tools are as battle-tested as OpenAI's offerings. GPT-Engineer, for instance, works but often the generated code might require manual fixes for edge cases. It can run basic executions, but ensuring it covers all scenarios of a `CPAEngine` might require extending its test routine. An error in our custom pipeline (like mis-specifying a prompt or a tool) could lead to confusing AI behavior that we have to troubleshoot. This approach likely needs multiple iterations and a strong validation suite around it to reach the confidence level of, say, Codex's agent.

- *Performance and resource constraints:* Running large models like Code Llama 70B or StarCoder locally demands significant resources (multiple GPUs or high-memory instances). If we cannot afford that, we might use a smaller model (which may produce lower-quality code or require more back-and-forth prompting). Alternatively, we might still call an API for a model like Claude or an OpenAI model via OpenRouter. That reintroduces some cost, but possibly less than using the full Codex agent service. In any case, optimizing for performance (e.g., using 7B/13B models for quick drafts, then a larger model for final pass) adds complexity to our pipeline.

- *Lack of editor integration out-of-the-box:* Unlike Copilot or Cursor, most open toolchain solutions are CLI or notebook-based. GPT-Engineer, for example, runs in a terminal and outputs files. Integrating this into VS Code would require either an extension UI or just instructing developers to run it separately. It's an extra step compared to in-IDE suggestions. We can build hooks (maybe a VS Code task that triggers our generator script), but it's not as naturally interactive unless we put effort into it.

Cost & Complexity: Using open-source tools is **license-free**, but not necessarily free in practice: there's the *opportunity cost* of development time and potentially the *infrastructure cost* of running models. If we go with GPT-Engineer or aider, the cost is mainly the OpenAI API usage (if they default to GPT-4) – for instance, one full generation might cost on the order of <$1 of API calls, and we can try cheaper models or limit tokens to reduce that. Fine-tuning our own model could incur a one-time cost in the thousands of dollars (cloud GPU time) plus maintenance, which is only justified if we expect heavy reuse. The **complexity** is the highest among the options discussed, because we're essentially taking on the task of orchestrating the AI ourselves. However, this complexity comes with the benefit of ultimate flexibility.

Feasibility for `CPAEngine`: It is feasible, but success will depend on our team's AI expertise and the refinement of the process. GPT-Engineer *can* generate non-trivial codebases from a prompt – for example, you could write "A Python program that listens to a Redis queue for tasks, uses OpenAI's API to analyze text, and returns structured results" and GPT-Engineer would attempt to create that. It might produce a working draft of `engine.py` plus maybe some agent logic file. In fact, GPT-Engineer emphasizes letting the AI *execute code* during generation [24], which means we could incorporate our real runtime (Docker + Redis) in the loop for validation. The feasibility of fine-tuning is more uncertain – with few examples, the model might not generalize well. A more practical approach could be to use prompt patterns (few-shot learning) where we provide a template `engine.py` (from another domain) in the prompt, then ask for a new one for the new domain. Many open LLMs can follow that with good results. With an open model scoring ~50% on HumanEval [26], it might require some manual fixes, but those can be caught in testing. In summary, this DIY approach **can produce a working engine** and gives us full ownership of the solution, but it likely requires the most up-front work to get to the same level of reliability as other options.

Selected Strategy

Recommendation: After evaluating all options, the **best strategy** is to combine the robust *autonomy of an AI agent* with the *usability of a VS Code extension*, using primarily open-source or self-hosted components. In practice, this means leveraging an **open agent toolchain (Option 4)** to generate and verify the `engine.py` logic, while providing a **VS Code integration (Option 2)** for developers to easily invoke and interact with the generation process. This hybrid approach capitalizes on reliability – the agent can run tests

and iterate, yielding higher confidence code – and on seamless integration – developers can trigger generation and see results in their familiar environment.

We choose this over directly using OpenAI's Codex service because of **control and cost**. While Codex (Option 1) is extremely powerful [3], relying on a closed platform for core development work raises dependency risks and could incur high ongoing costs. By using open frameworks (like GPT-Engineer's pipeline or the Cline VS Code agent extension) with our own prompt tuning, we approximate Codex's capabilities without lock-in. We also avoid solely relying on Copilot; Copilot is great for quick suggestions but lacks the iterative verification loop. Instead, our extension will call a backend process (our "Vibe-code engine generator") which uses an LLM in an *agentic fashion* – planning, code generation, execution, and refinement – then returns the final code to VS Code.

Why not only Cursor/Windsurf? Those tools indeed offer inspiration with their agent modes, but adopting them wholesale (Option 3) means less customizability for our specific workflow and no easy way to integrate into CI pipelines. Our recommended strategy can be run headlessly (e.g. as an NPM or Python CLI in CI) as well as via VS Code, ensuring that generated code can be automatically tested and even regenerated as part of a pipeline if needed. Windsurf/Cursor are excellent for individual productivity, but for a maintainable team solution, we prefer an open and scriptable agent approach.

Meeting the requirements: The chosen strategy will **generate reliable** `engine.py` **code** by using an agent that actually runs the code in a Dockerized test – this catches issues the first time. It offers a **VS Code command** to invoke generation, making it easy to use for developers during implementation of new features. It emphasizes **maintainability** by using open-source tooling that we can evolve (for example, updating the prompt or rules when our coding standards change, or upgrading the model). Deployability is handled by ensuring the output conforms to our project and container standards, and by integrating generation and test steps into **GitHub CI/CD** (e.g. a PR could include a regenerated engine, and CI will run it in Docker to confirm it processes tasks correctly). In sum, this strategy is a balanced, future-proof approach that addresses both the *technical robustness* and *developer experience* aspects of the Vibe-code App.

Implementation Plan

Below is a step-by-step plan to implement the Vibe-code App according to the selected hybrid strategy. We break it into phases – **environment setup, core generator development, VS Code extension, Docker/ Redis integration,** and **CI/CD pipeline** – to ensure systematic progress.

Phase 1: Development Environment Setup (Week 1)
1. **Set up AI model access:** Decide on the LLM backend. For initial development, use OpenAI's API (GPT-4) for convenience, as it offers high quality. Acquire API keys and set up a secure way to use them in development (e.g. environment variables). Simultaneously, prepare the ground for later integrating an open-source model (like installing local Code Llama 34B or ensuring we have access to an Anthropic API for Claude) to test alternatives.
2. **Prepare repository structure:** Create a new repository (or subfolder in c0engines) for the Vibe-code generator. Organize it with a clear structure, for example:
- `generator/` – Python or Node scripts that implement the AI agent workflow.
- `vscode-extension/` – the VS Code extension code (if using a separate extension).
- `prompts/` – any prompt templates or few-shot examples to be used by the generator.

- `test/` – sample tasks and maybe expected outputs for validation.

Ensure the repo has appropriate configuration (linters, etc.) since we'll be generating code within it. If using Python for the agent, set up a virtualenv and install needed libraries (OpenAI/Anthropic SDKs, LangChain if used, etc.).

3. **Baseline** `CPAEngine` **template:** Import or write a **template engine class** that the generator can use as a starting point. For example, create a file `prompts/engine_skeleton.py` containing the basic structure of a CPAEngine: the class definition, method signatures (`analysis, plan, run, summarize, recommend, design` returning a domain.Request), and perhaps a docstring explaining the engine's purpose. This will be provided to the AI as a guideline. Including an actual simple implementation (like a no-op engine that just marks tasks as success) could be useful as a reference example [27] [28].

4. **Redis + Docker setup:** Prepare a local Docker environment for testing generated engines. Write a basic `docker-compose.yml` that can bring up a Redis service and a container for running the engine. For example: one service for `redis:latest` and one for a Python container that mounts the generated code. This will allow the agent to spin up the environment to test the engine code. Also, create a small Python test script `test_task.py` that pushes a dummy task into Redis and invokes the engine (this script will be run inside the container to simulate usage).

Phase 2: Core AI Orchestration Logic (Week 2–3)

5. **Implement generation pipeline (CLI):** Develop a script (say `generate_engine.py`) that is the heart of the code generator. This script will:

a. **Read user input** describing the desired engine functionality. This could be a prompt file or command-line argument (for integration with VS Code, we'll pass the spec in). e.g. *"Engine should accept an invoice JSON, use an AI agent to classify VAT information (like VATDecisionMaker), and output a summary."*

b. **Prepare context for LLM:** Load the `engine_skeleton.py` template and the `IEngine` interface definition (from the c0engines project) into the prompt. Also include relevant domain model info if needed (like the shape of `domain.Request` or any constants). The prompt to the LLM will likely be a system message like: *"You are an AI that writes Python code for a CPAEngine class. Here is the interface and a template:"* followed by the code of the interface and skeleton, then a user message: *"Fill in the implementation so that it does X, Y, Z."*

c. **Call LLM to draft code:** Invoke the LLM API with the prompt and get the initial `engine.py` implementation. Save this to a temp file.

d. **Static checks:** Optionally, run a quick static analysis (linters or even just `python -m py_compile`) on the generated code to catch any syntax errors. If found, you can feed them back to the LLM in a second round: *"The code you wrote has a syntax error at line 50: ... Please fix it."* This is a lightweight way to handle trivial mistakes.

e. **Dynamic test run:** Now the crucial part – **Docker test execution**. Use Python's `subprocess` or a Docker SDK to build a container image that contains the generated engine and all necessary dependencies (we may mount the project directory into a generic Python image for speed). Then run the container with our test script. For example:

```
docker build -t vibe-engine-test -f Dockerfile.test .
docker run --rm --network=host vibe-engine-test python test_task.py
```

The `test_task.py` should instantiate the CPAEngine (the generated code) and simulate a task coming from Redis (or we can directly call the `run()` method with a fabricated request object). Capture the output or any exception.

f. **Evaluate test outcome:** If the engine runs successfully and produces a result, great – we consider this a pass. If there's an error (exception trace, or logical failure like it didn't produce the expected output format), feed that information back into the LLM. For example, if an exception occurred, send the LLM a message: *"The following error occurred when running the code: {traceback}. Please fix the code to address this."* This lets the LLM perform an iteration. We loop back to step (c) with this feedback. This loop can be bounded (e.g., max 3 iterations to avoid infinite loops).
g. **Finalize output:** Once the code passes the test (or if it reaches iteration limit), output the final `engine.py`. If multiple iterations occurred, ensure the final file is cohesive (the LLM should have been editing the file in each step – tools like GPT-Engineer handle multi-step refinement inherently).
This implementation might utilize an existing library: for instance, LangChain's `AgentExecutor` could manage some of these steps, or we could incorporate GPT-Engineer's own loop by providing our test as the execution step. Given GPT-Engineer already *"writes and executes the code"* [24], we might integrate our Docker test into its workflow. This saves time versus coding the loop from scratch.
6. **Incorporate multi-agent orchestration (if needed):** If the use-case expects the engine to coordinate multiple AI agents, ensure our prompting and testing reflect that. For example, the engine might call out to an OpenAI API or some `AgentManager` class. Since we can't actually hit OpenAI in the test (we want to keep it self-contained), consider mocking agent calls in tests or providing a dummy implementation for the agent (perhaps the generated code can use a fake response for testing). Another approach is to allow limited internet in the container and actually call a real AI API during test – effective but introduces external dependency in testing. Designing a smart way to validate agent orchestration (maybe by checking that certain methods are called in sequence, or the output structure is correct) will be part of this step.
7. **Validate on a simple example:** As a dry run, use a simple intention (like *"Engine that reads a number from Redis and returns its square."*). Run `generate_engine.py` and verify it indeed produces a working `engine.py` that, when run in Docker, squares a number. Adjust the prompt or code as necessary based on this test until the pipeline is stable. This ensures the core logic is sound before integration with VS Code.

Phase 3: VS Code Extension Development (Week 4)
8. **Scaffold the extension:** Use VS Code's Yeoman generator to create a new extension (`yo code` for TypeScript). Name it something like "Vibe-code Generator". In `package.json`, define a command e.g. `"vibecode.generateEngine"` that triggers the generation. Also set up an input method for the spec – perhaps a simple input box or a multiline input (quickPick or a temporary editor) where the user enters the functional description.
9. **Connect to generator backend:** There are a couple of ways to execute the generation from the extension:
- Easiest: have the extension **shell out** to our CLI script. For example, when the command runs, it saves the user prompt to a temp file and calls `python generate_engine.py --spec-file prompt.txt --output-file CPA/engine.py`. This requires the developer's environment to have Python and Docker available. The extension can then open the generated `engine.py` in the editor.
- Alternatively: call an API. We could wrap the generation logic in a local web server or have it listening on a port (perhaps the extension could spawn the Python script as a subprocess and communicate via stdout). This is more complex but can provide real-time feedback. A simpler variant: the extension shows the output log in an output channel. For instance, as the generator runs, stream the logs (which include progress or any AI messages) to a VS Code output pane so the user sees "Drafting code... Running test... Test failed, retrying... Test passed.".
For now, implementing the simpler approach (shell out and then show result) is fine.
10. **Provide configuration:** In the extension settings, allow configuring things like which model to use (OpenAI or local), API keys, or toggling iterative mode. We can store an OpenAI API key in VS Code's secure

storage if needed, or instruct users to have it in env vars (the extension can read env). Also allow the user to set the Docker context path or any special commands if their setup differs. Keep these configurable to make the tool flexible.
11. **Testing the extension:** Run the extension in VS Code's Extension Host mode. Try the command with a sample prompt. Ensure that after the command: - The user is prompted for input (or perhaps we detect if there's a `prompt.md` file open that we use as spec). - The extension calls the generator and waits. If using a subprocess, make sure to handle timeouts or errors (if the script crashes, capture that and show to user). - The `CPA/engine.py` file is created/updated in the workspace and automatically opened for the user. Possibly also focus a diff view if it existed before, so they can review changes. - The output channel (or status bar) indicates success or failure. For example, " Engine generated successfully and passed test" or "⚠ Generated engine, *but* test failed – please check and fix errors." In case of test failures that the AI couldn't fix in iterations, we leave it to the developer to manually intervene (with perhaps Copilot help). Ensure the extension handles edge cases like user canceling input or the `generator` script not found (give a clear error like "Please install the Vibe-code generator CLI"). Documentation can be added in the extension's README explaining prerequisites (Docker, Python, etc.).

Phase 4: Docker & Redis Integration Details (Week 5)
12. **Dockerize the whole flow (optional):** To make the generator itself portable, we can provide a Dockerfile so that the generation can run inside a container (with access to Docker socket to spin up test containers – possibly mounting `/var/run/docker.sock`). This would allow running the Vibe-code App in environments outside VS Code too, e.g. a CI server or a web service. This step might involve writing a Dockerfile that includes Python, the open-source model or OpenAI CLI, and our script. We can defer this if not immediately needed, but it's good to plan for deployment.
13. **Ensure Redis queue handling in generated code:** The `engine.py` we generate should properly connect to Redis and listen/publish to the correct channels or lists. To help the AI, provide a utility or guideline in context. For example, perhaps in `engine_skeleton.py` include pseudo-code:

```
def run(self) -> domain.Request:
    # Pseudocode: pop a task from Redis queue, process it, push result to
response queue.
    ...
```

So the AI will flesh that out. We should also give it the Redis connection details (maybe via environment variables or constants in our project that it can use). For instance, a constant `REDIS_URL` could be in context so the AI uses that. Testing this is crucial: in `test_task.py`, we can actually push a test message in Redis and see if the engine picks it up. If not using threading/async in engine, perhaps the test will directly call `run()` method instead of full queue operation. In any case, verify that the logic to interface with Redis is correct and doesn't block indefinitely (for testing, maybe we configure a non-blocking or a timeout).
14. **AI agents within engine:** For orchestrating *AI agents inside* the engine (the engine likely calls external AI APIs or uses something like `autogen.ChatAgent` as in the example code [29] [30]), ensure the generator knows how to include that. We might need to supply the import and usage pattern of our agent classes (like `VATDecisionMakerSystem` from the example [31]) in the context. Possibly maintain a library of common "skills" or agent invocation patterns that the AI can insert. The more specific we get, the easier for the AI to produce correct code. Over time, as we generate multiple engines for different tasks, we can build a **library of examples** which can be fed as few-shot examples to improve quality. For now, implement

support for at least one agent: e.g., if user says “use an LLM agent to extract keywords,” the generated code could utilize an OpenAI API call. Have the test verify that path (maybe by monkeypatching the API call to return a dummy result so the engine can finish).
15. **Logging and error handling:** Update the generator to encourage best practices in the output code – e.g., using our project’s logging (`logger`) as seen in the existing code [28], handling exceptions gracefully (returning a Request with error status on exception [32], etc.). We might include these as requirements in the prompt or even as part of the skeleton code (an empty try/except around the main logic, for instance). Since these touches improve maintainability, it’s worth enforcing them. The test can indirectly check this (for example, if an error is thrown, ensure the engine catches it and returns a proper formatted error, not crash).

Phase 5: CI/CD and Pipeline Integration (Week 6)
16. **GitHub Actions – Continuous Integration:** Create a GitHub Actions workflow for the c0engines repository (or wherever appropriate) to automate testing and possibly generation. Key steps:
- **Build Docker image** for the engine and run unit/integration tests. For example, have a job that builds the Docker image containing the latest `CPAEngine` and runs it against some sample tasks (similar to our local test but in CI). This will catch if a generated engine fails in a fresh environment.
- **Lint/format check** on generated code: to ensure the AI output adheres to style (we can run `flake8` or `black` in CI and fail if issues — the generator should already output formatted code if we prompt it to).
- **Security scan:** Since the engine runs untrusted prompts through AI agents, consider a static analysis or rules to ensure no obviously dangerous operations are present in the generated code (especially if the AI is used to generate it, we want to be sure it didn’t slip in something odd). A tool like Bandit could be run in CI on `engine.py`.
17. **GitHub Actions – Pipeline for generation (optional):** We can set up a workflow that, on a trigger (maybe a manual dispatch or when pushing a special branch), runs the Vibe-code generator to create/update `engine.py`. This could be useful for non-developers to propose a new engine by just editing a YAML or prompt in the repo and letting CI do the rest. For example, one could commit a file `new_engine_spec.txt` and the action would pick it up, run `generate_engine.py` with that spec, and commit the result back (perhaps via a PR). This is an advanced scenario, but it shows how the pieces (generator CLI + Docker) enable automation.
18. **Continuous Deployment:** If the engines are meant to be deployed as Docker containers (very likely), integrate the generation with the Docker build pipeline. For instance, ensure that whenever `engine.py` changes on the main branch, a new Docker image is built and pushed to a registry. This could be achieved by a simple change-detection in GitHub Actions or using Docker Hub's autobuild triggers. Since the engine uses Redis, you might also compose them together in a release. The key is that deployment always uses a tested engine version (which our CI ensured passes tests).
19. **Monitoring & Feedback:** Once deployed, it’s wise to monitor the performance of generated engines in staging before production. If possible, instrument the generated code to report metrics (e.g., time taken per task, error rates). This data can feed back into refining the generator prompts or logic. For instance, if we notice that certain patterns cause slow performance, we can adjust the generation to use a more efficient approach next time. This step is ongoing beyond initial implementation but is crucial for maintainability of the AI-generated code.

Phase 6: Validation & Testing Strategy (Week 7)
20. **Unit tests for generator:** Although the generator’s output is code, we should write tests for the generator itself. For example, given a fixed prompt and using a stubbed LLM (maybe a recorded response), does the pipeline correctly enter the loop, detect a failure, and iterate? We can simulate an LLM by using a simpler model or a fake that returns preset code on first call and improved code on second call. This

ensures our logic around LLM integration is sound.
21. **End-to-end test:** Take an example spec, run the whole extension or CLI as a user would, and then run the resulting engine in a live environment to see it produces the expected result. This can be semi-automated. For instance, as part of CI, we could have a test that uses our generator with a seed (perhaps use OpenAI's cheaper model to reduce cost during CI) and then verify the end result. If the AI output is nondeterministic, this is tricky; but we could lock the model to a deterministic setting (temperature 0) for testing a trivial spec to get a consistent result to assert on.
22. **User acceptance testing:** Have a developer team member (or the target user) use the VS Code extension to generate an engine for a realistic scenario. Gather feedback: Was the process intuitive? Did the code require a lot of manual fixes? Use this to refine prompts or extension UI. Possibly add convenience features to the extension, like previewing the plan the AI comes up with before code generation, or a command to regenerate only a specific method.
23. **Documentation & Examples:** Finalize documentation in both the repository and the VS Code extension README. Provide usage examples, e.g.: "To generate a new CPAEngine, open the Command Palette and run 'Vibe-code: Generate Engine'. Enter a description like: *Engine that reads invoices from a folder and uses AI to classify VAT for each.* The tool will create `CPA/engine.py` with the implementation. Review the code and run tests (we recommend `make test-engine`). If something looks off, you can tweak the prompt or the code and run the command again." Include troubleshooting tips (like ensuring Docker is running, etc.). This will help others (or future us) to effectively use and maintain the app.

By following these steps, we will incrementally build the Vibe-code App from a concept to a functional product. Each phase ensures that we have a working foundation before moving to the next, thereby mitigating risk. The result will be a system where, with a single command, a developer can generate a fully-fledged `engine.py` orchestrator that is immediately tested to be working in a Docker container environment – accelerating our development of AI agent orchestration logic significantly.

Next Steps Checklist (Timeline & Milestones)

- **Week 1: Project Setup & Research**
- [ ] Set up repository and development environment (AI API keys, base files, Docker Redis environment).
- [ ] Compile interface and template code for `CPAEngine` to guide the AI.
- [ ] (**Milestone:** Basic repo structure in place, able to run a dummy `generate_engine.py` that perhaps just prints "Hello AI").

- **Week 2: Prototype Generation Pipeline**

- [ ] Implement initial version of `generate_engine.py` that calls OpenAI API to fill in a simple CPAEngine skeleton (no iteration yet).
- [ ] Test with a trivial spec and manually inspect output.
- [ ] Add one-pass Docker execution test of output.

- [ ] (**Milestone:** Given a simple prompt, the tool generates an engine and runs it in Docker, even if it fails – we have the loop framework ready).

- **Week 3: Refine Agent Loop & Reliability**

- [ ] Implement iterative loop: detect runtime errors, feed back to LLM for fixes.
- [ ] Tune prompts for quality (e.g., ensure it knows to catch exceptions and log, etc.).
- [ ] Try a realistic scenario (maybe the invoice processing engine) and get a working result after some iterations.
- [ ] Experiment with an open-source model (if available) to gauge output differences.

- [ ] (**Milestone:** CLI tool consistently produces a working engine for at least one non-trivial use-case).

- **Week 4: VS Code Extension Integration**

- [ ] Scaffold VS Code extension and implement command to invoke generation.
- [ ] Integrate with the CLI (test on different OS for compatibility of spawning Docker, etc.).
- [ ] Add user input handling and output display in VS Code.
- [ ] Internal testing of extension inside VS Code (e.g., on a sample project clone).

- [ ] (**Milestone:** Vibe-code VS Code extension can generate an engine with a given prompt and open the file in the editor automatically).

- **Week 5: End-to-End Testing & Improvements**

- [ ] Write automated tests for generator (mock LLM to test loop logic).
- [ ] Run the entire pipeline on multiple example prompts to see if engines run correctly (cover different agent orchestration patterns).
- [ ] Improve prompt templates or extension UI based on test outcomes (e.g., if AI misses something consistently, add that to skeleton or prompt).
- [ ] Incorporate feedback logs so that if generation fails in extension, the user sees why (perhaps show the error trace or LLM response).

- [ ] (**Milestone:** High confidence in generation reliability; minimal manual fixes needed for new engines).

- **Week 6: CI/CD and Deployment Setup**

- [ ] Configure GitHub Actions for testing generated code (maybe run generation on a fixed prompt nightly as a regression test).
- [ ] Set up build pipeline to containerize the engine and run integration tests (ensuring future changes to generator or templates keep engines working).
- [ ] If applicable, set up a trigger or script for regenerating engines via CI for certain updates (perhaps when domain models change, etc.).
- [ ] Dockerize the generator itself for easy execution in CI or other environments.

- [ ] (**Milestone:** CI pipeline passes: it can generate an engine and run its tests in an automated fashion).

- **Week 7: Documentation & Team Training**

- [ ] Write comprehensive docs: how the generator works, how to use the VS Code extension, how to interpret results.

- [ ] Include example prompts and their outputs in the docs for reference.
- [ ] Hold a demo session with the team to introduce the new tool, generating an engine live to show usage.
- [ ] Collect developer feedback for any usability issues or desired features (e.g., option to choose model or to only regenerate a certain method).
- [ ] Finalize any changes from feedback.
- [ ] (**Milestone:** V1 of Vibe-code App released internally – developers are equipped to use it and it's integrated into our development workflow).

By following this timeline, we aim to have a functional and tested Vibe-code App in about 6–7 weeks. This delivers an AI-driven code generation capability that should significantly speed up developing new `CPAEngine` implementations, while ensuring those implementations are correct, maintainable, and easy to integrate into our existing systems [4] [24]. The result is a cutting-edge blend of AI and software engineering practices, positioning our team at the forefront of AI-assisted development.

---

[1] [2] [3] [4] OpenAI Codex: Transforming Software Development with AI Agents - DevOps.com
https://devops.com/openai-codex-transforming-software-development-with-ai-agents/

[5] [6] [7] [8] [9] GitHub - openai/codex: Lightweight coding agent that runs in your terminal
https://github.com/openai/codex

[10] [11] GitHub Copilot · Your AI pair programmer · GitHub
https://github.com/features/copilot

[12] Creating an OpenAI powered Writing Assistant for VS Code
https://rajeev.dev/creating-an-openai-powered-writing-assistant-vs-code-extension

[13] [14] [15] [16] [17] [18] [19] [20] [21] Windsurf vs Cursor: which is the better AI code editor?
https://www.builder.io/blog/windsurf-vs-cursor

[22] cline/cline - GitHub
https://github.com/cline/cline

[23] Cursor AI: The AI-powered code editor changing the game - Daily.dev
https://daily.dev/blog/cursor-ai-everything-you-should-know-about-the-new-ai-code-editor-in-one-place

[24] [25] GitHub - AntonOsika/gpt-engineer: CLI platform to experiment with codegen. Precursor to: https://lovable.dev
https://github.com/AntonOsika/gpt-engineer

[26] Paper page - Code Llama: Open Foundation Models for Code
https://huggingface.co/papers/2308.12950

[27] [28] [29] [30] [31] [32] engine.py
https://github.com/1010836/c0engines/blob/58498e71069c6f196650bd36a7b9ceeed3776521/src/cegid_pos_cpa/CPA/engine.py