# Internal Execution Plan: AI-Powered Python Security & Performance Scanner

## Introduction and Purpose

This hackathon project aims to build an **AI-powered security and performance scanner** for Python backend codebases. The goal is to automatically review Python code for common vulnerabilities and inefficiencies, using advanced language models to emulate an expert code auditor. Such a tool is extremely useful for development teams because it can **catch security flaws and performance bottlenecks early**, helping to prevent costly issues in production. By leveraging state-of-the-art AI, the scanner goes beyond simple linters or static analyzers – it understands code context and can detect subtle problems.

**Why leverage Cerebras?** The project takes advantage of Cerebras’s high-performance AI capabilities to enable fast, on-the-fly analysis. Cerebras Systems produce wafer-scale AI hardware that can run large language models with **high throughput and low latency**, which is ideal for a real-time code scanning tool. In our approach, we use **GPT-4.5 for heavyweight prompt engineering** (done before the hackathon) and then utilize a **Cerebras-hosted Llama 4 model for rapid inference** during the hackathon event. This combination allows us to pre-generate complex scanning logic with a powerful model, and then execute those scans quickly at hackathon time. The end result will be a tool that, within the limited hackathon timeframe, can **analyze a Python codebase and produce a report of security vulnerabilities and performance issues**, showcasing the efficiency and power of the Cerebras + LLM solution.

**Specialties about Cerebras:**  
Cerebras Systems specializes in developing advanced computing solutions tailored for artificial intelligence (AI) and deep learning applications. Their flagship innovation is the **Wafer-Scale Engine (WSE)**, recognized as the world's largest computer chip. The latest iteration, **WSE-3**, measures approximately 21 centimeters per side and integrates 4 trillion transistors, providing unparalleled computational power. ​

**Key Capabilities of Cerebras Systems:**

1. **High-Performance AI Inference:**
   * Cerebras offers an AI inference solution that delivers up to 20 times the performance of traditional GPU-based systems at a fraction of the cost. For instance, their platform achieves processing speeds of 1,800 tokens per second for Llama 3.1 8B models and 450 tokens per second for Llama 3.1 70B models. ​
2. **Scalability and Efficiency:**
   * The WSE's design consolidates multiple processing elements onto a single wafer, enhancing data transfer speeds and reducing energy consumption. This architecture enables the handling of complex AI models and large datasets more effectively than traditional multi-chip configurations.
3. **Support for Large Language Models (LLMs):**
   * Cerebras' infrastructure is optimized for training and deploying large language models. Their systems have been utilized to train models with billions of parameters, facilitating advancements in natural language processing and related fields. ​
4. **Enhanced Reasoning Capabilities:**
   * The introduction of the **Cerebras Planning and Optimization (CePO)** tool has significantly improved the reasoning abilities of models like Llama 3.3 70B, enabling them to outperform larger counterparts in various benchmarks. ​
5. **Extensive AI Infrastructure:**
   * Cerebras is expanding its AI infrastructure by establishing multiple data centers across North America and Europe. This expansion aims to increase inference capacity to over 40 million tokens per second, addressing the growing demand for AI processing power. ​

In summary, Cerebras Systems offers cutting-edge solutions that enhance the performance, scalability, and efficiency of AI and deep learning tasks, positioning itself as a significant player in the AI hardware industry.

**Specialties about Llama 4:**   
Meta's Llama 4 series introduces several notable advancements in artificial intelligence models:

* **1. Mixture-of-Experts (MoE) Architecture**: Llama 4 employs an MoE design, activating only the necessary model components for specific tasks. This approach enhances computational efficiency and performance, allowing for more effective processing of complex inputs.
* **2. Multimodal Capabilities**: These models are designed to handle various data types, including text, images, video, and audio. This versatility enables applications across diverse domains, from conversational AI to image recognition.
* **3. Enhanced Context Windows**: Llama 4 models feature extended context windows, with Llama 4 Scout offering a 10-million-token capacity. This allows the models to consider more extensive input data, leading to more coherent and contextually relevant outputs.
* **4. Open-Weight Models with Licensing Considerations**: While Meta promotes Llama 4 as open-weight, entities with over 700 million monthly active users must obtain permission for commercial use. This balances open access with proprietary control. citeturn0news18
* **5. Reduced Political Bias and Improved Handling of Contentious Topics**: Llama 4 demonstrates a decreased tendency to avoid politically or socially charged questions, declining less than 2% of such prompts compared to 7% by its predecessor. Additionally, it exhibits reduced political bias, aiming for balanced responses on sensitive issues.

## Tech Stack

Our implementation will integrate several technologies to achieve the desired functionality:

* **Python Backend**: The core scanner logic will be written in Python. Python is ideal since the codebase being analyzed is Python, and we can easily parse and manipulate Python code (using libraries like ast or regex as needed). Python also allows quick integration with machine learning libraries and APIs for GPT and Cerebras.
* **GPT-4.5** (Prompt Engineering Phase): We will use the GPT-4.5 model (presumably via OpenAI API or a research preview) before the hackathon to assist in crafting effective prompts and scanning templates. GPT-4.5’s superior understanding and generation abilities will help formulate the questions/instructions that will later be fed to the scanning model. This is essentially an offline/Pre-computation step using a very powerful model to guide our approach.
* **Cerebras Inference API with Llama 4**: During the hackathon, the actual code scanning will be performed by a **Llama 4** model (the latest generation of Meta’s Llama series) served on a Cerebras system. We assume access to a Cerebras AI accelerator and its inference API, which allows sending prompts (with code) to the Llama 4 model and getting results quickly. The Cerebras hardware ensures that even though Llama 4 might be a large model, we can get responses in near real-time, which is crucial for scanning possibly hundreds of lines of code within the hackathon demo.
* **Frontend for Results**: To present findings, we plan to build a simple user interface. Given the time constraints of a hackathon, a lightweight framework like **Streamlit** (for a quick dashboard) or a minimal **Flask** web app can be used. This UI will list detected issues categorized by type and severity, and possibly highlight problematic code segments. The focus is on clarity of results for demo purposes.
* **Supporting Tools**: We will use standard Python libraries for ancillary tasks – e.g., gitpython or pygit if pulling code from a repo, file system operations for code ingestion, and possibly a syntax highlighter for displaying code in the UI. If needed, we might incorporate a vulnerability database (for dependency checks) or use pip’s safety DB to flag vulnerable packages, etc. However, the primary analysis of code logic is done via the LLM.

Overall, this tech stack balances **power and speed**: GPT-4.5 provides the intelligence in designing the solution, and Cerebras+Llama4 provides the muscle to execute it quickly. Python glue code ties everything together and presents the results in a human-friendly format.

## Prompting Strategy

Our scanning approach uses a **two-layer prompting strategy** to effectively harness the strengths of different AI models:

1. **Prompt Pre-Generation with GPT-4.5 (Offline Layer)** – Before the hackathon event, we will invest time in engineering the prompts using GPT-4.5. In practice, this means using GPT-4.5 to generate and refine the questions or instructions that will later be asked to the Llama 4 model for scanning the code. We treat GPT-4.5 as a brainstorming and QA partner to develop the best possible prompts. For each category of issue (auth, injection, etc.), GPT-4.5 can help create a template like: “You are a security auditor. The following is a Python function. Analyze it for any input validation or injection vulnerabilities and explain any issues found.” We will iteratively improve these prompts by testing them (perhaps even asking GPT-4.5 to role-play as if it were the scanning model to see if the prompt yields the kind of answers we want). The outcome of this phase is a **toolkit of well-crafted prompt templates** for all the different types of checks we want to perform.
2. **Real-Time Scanning with Cerebras + Llama 4 (Execution Layer)** – During the hackathon, we switch to using the prepared prompts on the Llama 4 model running on Cerebras hardware. This is the execution layer where code is actually analyzed. The flow will be: for each piece of code (file, function, or code snippet), we feed it into the Llama 4 model along with the appropriate prompt template asking for specific issues. Because the prompts have been optimized in advance, Llama 4 can focus purely on inference, which is fast on the Cerebras system. We will use a **multi-round prompting approach** for thoroughness:
   * First, run the code through the model with each vulnerability category prompt (one by one or in parallel threads). Each prompt is tailored to find a particular class of issues. For example, one prompt specifically checks for authentication/credential problems, another for injection flaws, and so on.
   * Since we have multiple categories, the code might be analyzed multiple times (once per category) with different prompts. This ensures comprehensive coverage, as a single prompt asking for everything might cause the model to miss some details. The two-layer strategy shines here: the prompts are specific and pointed (thanks to GPT-4.5’s prep work), and Llama 4 can respond quickly to each specific query.
   * If needed, we can do a **follow-up round**: If the initial results from Llama 4 are unclear or if certain high-risk sections of code are identified, we can prompt again with more detail or ask the model to double-check a certain function. For instance, if an output mentions a possible SQL injection in a function, we might run a second prompt on just that function asking “explain how an attacker might exploit this.” This ensures we have enough detail for the report and helps verify true positives vs false positives.
   * We will avoid extremely lengthy prompts or outputs to keep inference fast. Each prompt will focus on one aspect at a time, keeping the model’s task constrained and execution efficient.
3. **Few-Shot vs Zero-Shot**: Our prompt templates could include few-shot examples if it improves results, but given the length constraints (we might be feeding large code blocks), we will likely use **zero-shot instructions** or one-shot at most. GPT-4.5 can help determine if adding an example in the prompt helps Llama 4. If examples are used (e.g., a tiny snippet and its identified issue as a guide), they will be very short and generic, just to guide the model’s style of output (for instance, showing the format “Issue: X, Explanation: ..., Recommendation: ...”).
4. **Output Formatting**: Part of the prompting strategy is also to dictate how Llama 4 should format its findings. We will craft the prompts to request answers in a structured format (for example: list each issue with a short title and description, or maybe in JSON for easy parsing). This will help later when we aggregate and display results. GPT-4.5 can assist in designing this output schema during the prep phase. During the hackathon, we’ll ensure each inference from Llama 4 follows the format (and if not, we might do a minor re-prompt like “please output in the requested format”).

By separating prompt creation and prompt execution between two models, we ensure that **each model is used to its strength**: GPT-4.5’s creativity and deep understanding to set up the logic, and Llama 4’s speed on Cerebras to apply it at scale. This two-layer prompting strategy is at the heart of our project’s feasibility and effectiveness.

## Detection Scope

The scanner will focus on **application-level vulnerabilities and performance issues** in Python backend code. Specifically, we will cover the following major categories, ensuring that for each category the scanner has dedicated checks:

* **Authentication & Credential Issues**: This covers flaws in how the application handles user authentication and secrets. The scanner will look for hard-coded credentials in the code (e.g. API keys, passwords), usage of weak password storage mechanisms (like storing plaintext passwords or using weak hashes), and insecure authentication logic. For instance, it will flag if it finds an admin password like "password123" in the source, or if JWT verification is skipped. It also checks for missing security steps (like not verifying a password’s hash correctly, or missing multi-factor checks if expected).
* **Input Validation & Injection**: This category deals with how the code handles external input and ensures it’s safe. The scanner will search for places where user input is directly used in commands or queries without proper sanitation. Typical issues include SQL Injection (building SQL strings directly with user input), OS command injection (os.system or subprocess calls with unsanitized input), LDAP or NoSQL injections, and general lack of input validation (e.g., not validating form fields, leading to unexpected behavior). The tool will flag patterns like string formatting that include user variables in database queries, usage of eval() on user input, or constructing file paths directly from user-provided data (possible path traversal).
* **Access Control Flaws**: These are issues related to authorization – making sure users can only do what they’re allowed to do. The scanner will try to identify routes or functions that should enforce permissions but perhaps don’t. For example, it will look for endpoints that don’t verify the user’s role or privilege before performing sensitive actions, or direct object reference issues (where an object ID is used from user input without checking ownership). Hard-coded role checks that might be bypassable, or missing checks on certain conditions (like assuming security on front-end without back-end verification) would also be caught. Essentially, anywhere the code fetches or manipulates data based on user-provided identifiers, the scanner ensures an access control check accompanies it.
* **Cryptography Misuse**: This focuses on how the code uses cryptographic functions and libraries. Common pitfalls include using outdated or insecure algorithms (like MD5 or SHA1 for security-critical hashing), hard-coding cryptographic keys or salts, not using encryption for sensitive data, or implementing cryptography incorrectly (like using a constant IV for CBC encryption, or not verifying SSL certificates in outbound connections). The scanner, using the LLM’s knowledge, will flag things like hashlib.md5(password) (weak hashing) or use of the random module for generating security tokens (which should use secrets module or similar). It will also highlight if encryption is used without integrity (missing HMAC) or if the code disables certificate verification in HTTPS requests (common in requests library via verify=False).
* **Dangerous Coding Patterns**: This is a broad category capturing code practices that are risky and often lead to vulnerabilities. The scanner will watch for use of Python functions like eval(), exec(), or pickle.load() on untrusted data – all of which can execute arbitrary code if misused. It also includes catching broad exceptions and ignoring them (which can hide security issues), use of debug backdoors (like a route that opens a shell), and other anti-patterns. Another example is using assert statements for enforcement (since running Python in optimized mode will skip asserts, which could bypass a security check). These patterns might not be a vulnerability by themselves, but they indicate a high risk and are worth flagging for review.
* **Logging and Error Handling**: Here we ensure the code properly handles errors and logs information without exposing sensitive data. The scanner will check that exceptions are caught and handled so that the application doesn’t crash or reveal stack traces to users. It will flag any instance of sensitive information being logged (e.g., logging full credit card numbers, passwords, or personal data). It also verifies that important security events (like failed login attempts or permission denied events) are logged, since lack of logging can hinder breach investigations. Essentially, the model will look for logger.debug() or print() statements that output secrets, or except: pass blocks that swallow errors, and raise those as issues.
* **Dependency Risks**: Modern applications rely on many third-party packages, and outdated or vulnerable dependencies are a major risk. The scanner will parse the project’s dependency list (such as a requirements.txt or Pipfile.lock) to find any known vulnerable versions (for example, using a version of Django with a known CVE). We can integrate a known vulnerabilities database or simply use GPT’s knowledge of notorious packages (for the hackathon MVP, we might hardcode a few examples of what to catch, like flagging if flask==0.12 is used because it’s old). Additionally, if the code itself uses import statements for modules that are deprecated or known to be problematic (like the old crypto library, or using yaml.load without SafeLoader), those will be highlighted. The focus is to inform the developer that “hey, library X at version Y has known issues, consider upgrading or patching it.”
* **Performance & Maintainability**: Apart from security, we also want to highlight performance anti-patterns and code quality issues that could make the backend slow or hard to maintain. The scanner will search for obvious inefficiencies like **nested loops on large data structures** (which could be O(n^2) or worse), doing blocking I/O or sleep calls on the main thread of a web server, or not using caching where it’s apparent it could help (for instance, repeatedly reading a file inside a loop). It also flags extremely complex or long functions which indicate maintainability issues (e.g., a single function spanning hundreds of lines, which is hard to read and test). Another aspect is memory usage patterns – like loading an entire huge file into memory instead of streaming. While these may not be outright “bugs,” they are important for a robust, scalable application and make a good value-add in the report. For the hackathon, we’ll likely manually identify a couple of performance patterns to detect (for example, using pandas for a simple task in a web service might be overkill, or lack of database indexing if we can see query patterns) and have the model flag them.

By covering the above categories, our scanner aims to address most of the OWASP Top 10 security risks (injection, auth, access control, etc.) as well as common Python-specific pitfalls. We are intentionally focusing on **application-level logic issues** rather than low-level issues (since Python is memory-safe, we don’t worry about buffer overflows, etc.). Each category corresponds to a set of checks we have prompt templates for, ensuring the LLM scrutinizes the code from all these angles. In summary, the scope is comprehensive enough to catch critical problems but also scoped to what’s achievable with LLM analysis within a short timeframe.

## Timeline and Tasks

To execute this project within the hackathon, we outline a clear timeline with tasks split between a **pre-hackathon preparation phase** and the **hackathon day execution phase**. This ensures that we maximize what we can get done during the fast-paced hackathon by doing as much upfront work as possible.

### Pre-Hackathon (GPT-4.5 Preparation Phase)

Prior to the hackathon (days or week before), the team will complete the following tasks using GPT-4.5 and other planning tools:

1. **Define Categories & Success Criteria**: Finalize the list of vulnerability/issue categories (as listed above) we want to cover. For each category, write down what specific issues within that category look like in code (e.g., for “Input Validation”, note patterns like usage of unsanitized input in SQL queries). Essentially, create a mini knowledge base of what to look for in Python code for each category.
2. **Prompt Template Generation**: Using GPT-4.5, generate initial prompt templates for each category. We will do this by literally asking GPT-4.5 to help: for example, “Help me craft a prompt that instructs an AI to find authentication and credential handling issues in a Python code snippet.” We’ll iterate on the wording. The output should be a prompt that we can use with another LLM (Llama 4) that clearly asks for issues in that category and maybe the format of answer. We’ll do this one category at a time.
3. **Refinement through Iteration**: For each prompt template generated, test it with GPT-4.5 or a smaller model to see how it performs. We might take a small piece of code with a known flaw and see if the prompt leads the model to catch it. For instance, take a snippet with a hard-coded password and see if the “Auth & Credential” prompt catches it. If it misses or the output is not well-structured, refine the prompt wording. (GPT-4.5 can also critique the prompt and suggest improvements). This is an iterative loop: prompt -> test -> refine.
4. **Organize Prompt Toolkit**: Once finalized, organize all the prompt templates and guidelines into a “prompt toolkit” – basically a document or JSON file where each category has an associated prompt string (and maybe some notes on how to use it). This will be packaged with our code so that the scanning script can easily retrieve the correct prompt for each type of check.
5. **Prepare the Infrastructure**: Before the hackathon, ensure we have access to the Cerebras API and the Llama 4 model. This may involve setting up credentials or environment to call the Cerebras service. We should test a simple query to the model (like a “hello world” prompt) to confirm we can get responses. Also, set up the skeleton of the Python project – create a repository, stub out a basic command-line or web app structure, and perhaps include a few sample code files for testing.
6. **Plan Demo Scenario** (if time permits): Think about how we will demonstrate the tool. We might prepare a small intentionally vulnerable Python project or identify a subset of an open source project that we can run the scanner on during the demo. Having this ready means on hackathon day we aren’t scrambling to find test data. (This step is optional but helpful for a smooth presentation.)

By completing these steps in advance, we will enter the hackathon with a clear plan, a set of ready-to-use prompts, and a configured environment. Essentially, the heavy “thinking and designing” work is done upfront, allowing us to code and integrate quickly during the competition.

### Hackathon Day (Execution Phase with Cerebras)

On the day of the hackathon, with preparation in hand, the focus is on coding the solution, integrating the AI model, and producing a polished output. The tasks for the hackathon day are as follows (in rough sequence):

1. **Morning: Setup and Code Ingestion** – Start by finalizing the development environment. Load the target Python codebase we intend to scan (this could be the project’s own code or a test codebase). Implement a module to **ingest code**: this might involve reading all .py files from a directory, or connecting to a GitHub repo. We’ll structure the code so we can iterate through modules or functions. Ensure that the Cerebras API endpoint and keys are accessible in our environment (perhaps set as environment variables).
2. **Implement Scanning Logic** – Using the prompt toolkit prepared earlier, write the code that sends each code snippet through the Llama 4 model with the appropriate prompt. Likely, we will create a function like scan\_code(file\_path) that reads a file, possibly breaks it into logical units (e.g., functions or classes) if it’s large, and for each unit runs the set of prompts. We must be mindful of token limits: if a file is too large, we might feed one function at a time to the model. At this stage, focus on getting responses from the model for each category. This involves calling the Cerebras inference API, waiting for the response, and capturing the output. We should also tag each result with what file and which category it belongs to.
3. **Multi-Round Inference** – Ensure that the process covers all categories. This could be done sequentially (iterate categories for each file) or in parallel if the API and time allow (maybe using threads or async calls to scan different files or categories simultaneously). After initial results are obtained, review them quickly:
   * If some obvious issues are flagged and need more detail, we can run a quick follow-up prompt. For example, if an output says “Potential SQL injection via user\_input in function get\_data,” we might run a second prompt specifically on get\_data asking for an explanation and fix recommendation, to enrich our results.
   * Also, watch out for false positives or irrelevant info. If the model output sometimes includes noise (not an issue), we might filter those out or mark them as low confidence. This could be done by simple keyword filtering or by confidence heuristics (if available).
4. **Aggregate and Organize Results** – As we get findings from the scanning logic, aggregate them into a unified report structure. For instance, create a Python object or data structure that holds all findings, organized by category and by file. This is where having a consistent output format helps (if we got JSON from the model, we can parse and merge easily). We will likely generate a summary such as:
   * For each file scanned, list all issues found, each with the category, description, and recommendation.
   * Or, for each category, list which files had issues of that type. Or both. For demo purposes, per-file listing is user-friendly (“here’s what was found in models.py, here’s what was in utils.py…”).
5. **User Interface Development** – Develop a simple interface to display the results. If using **Streamlit**, we can quickly make a page that has sections for each file or category, expanding to show details of each issue. If using **Flask** or another web framework, create an HTML page or template that iterates over the results data and prints out lists and perhaps some color-coding (e.g., red icon for high severity issues like injections, yellow for warnings like minor performance issues). The UI does not need to be extremely fancy, but it should be clean, readable, and organized as per categories and severity. We’ll also include a brief introduction on the page about what the tool is doing (for context in the demo).
6. **Testing the End-to-End Flow** – Once the scanning logic and UI are in place, run the tool end-to-end on the chosen sample codebase. This is to verify that:
   * The code ingestion correctly reads all intended files.
   * The Cerebras+Llama pipeline returns answers for each prompt without errors or excessive delay.
   * The results are correctly parsed and displayed in the UI.
   * There are no crashes or unhandled exceptions (e.g., if the model returns unexpected format). This testing might reveal last-minute bugs (for example, maybe one category’s prompt was too long and needs trimming, or a certain file had no issues and our code didn’t handle an empty result well). We’ll fix those quickly during the hackathon.
7. **Performance Tweaks (if needed)** – If we find the scanning is slow (maybe the codebase is large and we have many prompts), we might make minor optimizations. For instance, skip scanning certain files like migrations or settings that are less relevant, or reduce prompt size for very large functions (maybe analyze only parts of it). The Cerebras hardware should give us speed, but we must be mindful of how many total analyses we run. Since this is an internal plan, we note these potential adjustments, but we’ll gauge on the fly during the event.
8. **Final Polishing** – In the final stretch of the hackathon, we’ll polish the presentation of results and the overall project:
   * Double-check the content of the findings: ensure the wording is clear (the model’s output might be tweaked in prompt to be concise). If some issues are phrased awkwardly, we can post-process the text lightly (or adjust prompts once more) to make it clear for judges/users.
   * Add severity labels or prioritization in the output if not already present (e.g., mark injection issues as "High Risk", performance suggestions as "Info"). This can simply be done based on category mapping to severity.
   * Make sure every category of issue at least has one example in the output (if our test code doesn’t have one category, maybe we insert a synthetic small example to show the tool catching it, purely for demonstration completeness).
   * Clean up the code repository (remove debug prints, add comments, prepare to open source if intended).
   * Prepare a few talking points for the presentation/demo: for instance, highlight “This issue found here is something that could lead to a data breach, and our AI caught it within seconds” – such narrative points will make the demo impactful.

By following this timeline, we ensure that we **utilize the hackathon time effectively**, focusing on integration and showcasing results, because the conceptual heavy-lifting (what to check and how to prompt the AI) was done beforehand. Each team member can take ownership of some parts (one handles the AI integration, one the UI, one the testing and results polishing, etc., if team size allows) to work in parallel.

In conclusion, this execution plan provides a structured path to build a powerful AI-driven code scanner within a short timeframe. By carefully planning the prompt strategy with GPT-4.5 and harnessing Cerebras’s fast inference for Llama 4, our project will demonstrate how AI can significantly augment software security reviews and performance tuning, all in the spirit of an innovative hackathon project. The end deliverable will be an internal tool (with a possible future as a product) that can scan Python backend codebases and output a clear report of potential security vulnerabilities and performance issues, all accomplished in just one hackathon’s effort.