ChatGPT

Incident Sage – Product Requirements Document (MVP)

Overview

Incident Sage is an open-source tool for DevOps/SRE teams to automatically analyze application logs (primarily JSON-formatted) and detect potential incidents. It provides concise, AI-generated summaries of incidents identified in logs, helping engineers quickly understand what went wrong without manually parsing thousands of log lines [1] . The tool supports both a command-line interface (CLI) and a lightweight web-based UI for flexibility. All AI analysis runs **locally** – no cloud APIs are used – to ensure data privacy and enterprise trust (i.e. all log processing is 100% offline with no OpenAI or external dependencies [2]). This MVP focuses on core functionality and is scoped to be built in 4 days, covering only the essential features needed for incident detection and summarization.

Key Goals and Rationale:
- **Accelerate Incident Triage:** Enable engineers to get a quick summary of error patterns and likely causes from log files, useful for noisy log triage and post-mortems [1] .
- **Privacy-Preserving AI:** Use a locally hosted Large Language Model (LLM) (e.g. via llama.cpp or Ollama) for analysis, avoiding sending sensitive logs to any cloud service [3] .
- **Ease of Use:** Provide both CLI and Web UI options for convenience – CLI for quick terminal analysis (e.g. over SSH on a server) and Web UI for a more visual exploration.
- **Rapid Deployment:** Containerize the application for easy deployment on Docker or AWS ECS, aligning with common DevOps workflows.

Functional Requirements

1. CLI Tool

The CLI component of Incident Sage allows quick analysis of log files via the command line. It should support the following features:

- **Input Modes:** Accept log data via a file path (using a `--file <path>` flag) or via standard input (piped from another command). This enables usage like `cat app.log | incidentsage` as well as `incidentsage --file app.log`.
- **JSON Log Parsing:** Parse and handle JSON-formatted log entries. The tool must be able to detect JSON objects even if they are embedded in text lines (some log aggregators embed JSON in a single line). Non-JSON lines or malformed JSON should be handled gracefully (ignored with a warning or error message, but not crash the tool).
- **Incident Pattern Detection:** Scan the logs for **"incident-worthy" patterns** that likely indicate an issue:
- Error-level or high-severity log entries (e.g. lines with levels `ERROR` or `WARN`).

- *Optional (MVP if time permits):* Detect anomalous spikes in errors or log volume over short time windows. This could involve simple heuristics (e.g. X errors within Y minutes). (Advanced anomaly detection or machine learning-based spike detection is out of scope for the MVP, but design should allow adding this later.)
- **Incident Window Extraction:** When an incident pattern is found (e.g. a burst of errors or a critical error), extract the relevant log lines around that time frame. For example, gather a window of log entries before and after an error to provide context. The tool might collect a few minutes of logs or a certain number of lines surrounding the detected incident point.
- **AI Summary Generation:** For each incident detected, generate a short summary of the incident using a local LLM. The CLI should feed the relevant log excerpt to the LLM with a prompt asking for a concise summary and possible root cause. The prompt design will include clear instructions to the model (e.g. "Summarize the incident in 2-3 sentences, focusing on what happened and potential causes, without verbose output"). The LLM output should be limited to a brief summary (around 2-3 sentences per incident) to ensure it's concise.
- **Output:** Print the AI-generated incident summaries to the console, along with any key details like timestamps of incident occurrences or error codes. The CLI output should also include *suggested causes* if the LLM infers any (e.g. "database connection failure" or "out-of-memory error"), and possibly a reference to the first error log message of each incident. The output format could be plain text or markdown. For MVP, simple text with clear labeling is fine (e.g. "**Incident 1:** … summary …").
- **User Feedback & Responsiveness:** If analyzing a large log file, the CLI should remain responsive. Include a spinner or progress indicator while the LLM is processing large inputs [2] so the user knows the tool is working. If no incidents are found in the log, the tool should inform the user (e.g. "No incident-worthy events detected in the log").
- **Error Handling:** Provide clear error messages for common failure cases (e.g. "File not found," "Invalid JSON format at line X"). Malformed log lines should not stop the analysis; they can be skipped with a warning message so that one bad line doesn't break the whole summary process.

Usage Example (CLI):
A DevOps engineer can run Incident Sage on a log file after an outage: for instance, `incidentsage --file server_log.json`. The tool might detect a surge of ERROR logs around 12:05 UTC and an exception stack trace. It would extract those logs and use the LLM to output a summary such as: *"Around 12:05 UTC, multiple database connection errors occurred, causing request failures. The service could not reach the DB for 3 minutes, likely due to a network issue, and recovered afterwards."* This gives the engineer a quick insight without reading the entire log file. (If the engineer was on the server, they could similarly do `journalctl -u myapp.service -n 500 --no-pager | incidentsage` to summarize the last 500 lines of system log [4].)

2. Web UI

Incident Sage will also include a minimal web-based interface that runs from the same application (no separate server component beyond the binary). The Web UI provides a user-friendly way to upload and analyze log files:

- **Log File Upload:** Provide a simple webpage where users can upload a log file (through an HTML file input). Once uploaded, the backend will run the same analysis as the CLI on that file.
- **Trigger Analysis:** After upload, the user can click an "Analyze" button (or it can auto-run upon upload) to start the incident detection. The UI should indicate progress (e.g. a loading spinner or status message) while the logs are being processed by the LLM.

- **Incident Summary Display:** Once analysis is complete, display the results on the page. This includes:
- A concise **incident summary** generated by the AI for each incident found (each summary 2-3 sentences as above).
- The **top error traces or messages** associated with the incident. For example, the UI might show the error log lines (stack trace or error message) that triggered the incident detection, so the user can see the raw error details.
- **Log Exploration:** Provide a way for the user to explore the log context within the UI:
- At minimum, display the portion of the log around each incident (the same lines that were input to the summary). This could be a collapsible section or a scrollable text area showing the raw log lines before/after the incident trigger point.
- Optionally, allow the user to search within the log or jump to specific timestamps (nice-to-have if time permits).
- **Timeline Visualization (Optional):** If feasible, include a simple timeline or graph showing log intensity over time (e.g. a histogram of log entries per minute or an error count over time). This can help visualize where spikes or errors occur. However, this is not core to the MVP and should only be attempted if it can be done very simply (for example, a basic chart library to plot counts of ERROR logs by timestamp). If not implemented, it can be noted as a future enhancement.
- **No Authentication (MVP):** The Web UI for MVP will be intentionally simple and likely not secured by login (since it's meant for local or internal use). It will be a lightweight interface primarily for a single-user scenario. (In future, if used in multi-user environments, authentication and access control would be added – see Out of Scope.)
- **Responsive Design:** The UI should be usable from a desktop browser. It doesn't need a sophisticated design – just a clean layout to show the upload form and results. Focus on clarity: e.g. use headings or cards for each detected incident's summary and logs.

3. AI Integration (Local LLM)

A core feature of Incident Sage is the integration of a local Large Language Model to interpret and summarize log data. Requirements for the AI component include:

- **Local Model Usage:** The system **must not rely on external AI APIs**. It will use a locally running LLM, ensuring data never leaves the host. For example, the tool might use **llama.cpp** or **Ollama** to run a model like Llama-2 7B or Mistral 7B directly on the machine [5]. All inference is done on the local CPU (or GPU if available) within the container.
- **Model Performance:** The chosen model should be small and efficient enough to run within the resource constraints (see Non-functional Requirements) while still providing coherent summaries. Models on the order of 7 billion parameters (possibly quantized to around 4GB of memory) are anticipated as a good balance. For instance, an open-source 7B model can generate reasonable summaries and can be run on commodity hardware [6] [3].
- **Prompt Engineering:** Design the prompts fed to the LLM such that it understands the task. The prompt will likely include a brief instruction (e.g. *"You are an assistant that summarizes application log incidents. Given the following log excerpts, provide a brief summary of the incident in 2-3 sentences, mentioning the issue and possible cause."*) followed by the log excerpt. To keep context concise, the log excerpt may be truncated to a window around the incident (so we don't exceed model input limits if the log is large). The prompt should explicitly tell the model to be concise and not exceed a certain length (to avoid overly verbose output).
- **Output Control:** Implement safeguards on the AI output:

- **Length Limit:** The summary should ideally be no more than a few sentences (the model can be instructed, and we can also programmatically truncate if needed).
- **Relevance:** The model should focus only on the information in the logs. If the model output is not relevant or hallucinates information not present in the logs, the tool should flag or discard that output. (For MVP, we rely on prompt constraints and model selection to minimize this issue.)
- **Multi-Incident Handling:** If multiple distinct incident patterns are found in one log file, the system should run the model separately for each incident window. The output would then be a series of summaries (Incident 1, Incident 2, etc.). Ensure the prompt or code handles multiple segments sequentially without mixing them.
- **No Internet or Cloud Calls:** The runtime environment will not have the model call out to any cloud API. This means the model must be downloaded and packaged as part of the tool deployment (or downloaded at install time). The design should note the model used (and its open-source license) and provide instructions or scripts to obtain it. This approach is similar to other tools that bundle an OSS model for offline use [2] .

4. Deployment

The product must be easy to deploy in typical DevOps environments. Deployment-related requirements:

- **Docker Support:** Provide a Dockerfile that can build a container image containing the CLI tool and the web UI. Running the container should launch the Incident Sage service. For example, a user should be able to do `docker build -t incidentsage .` and then run `docker run -p 8080:8080 incidentsage` to use the tool (serving the web UI on port 8080, and CLI could be invoked inside the container as well). The container should include the local LLM model or have a mechanism to fetch it on first run.
- **ECS Compatibility:** Ensure the Docker image is compatible with AWS Elastic Container Service (ECS) deployment. This means:
- The container should log to stdout/stderr so that ECS can capture logs (which often go to CloudWatch).
- Provide documentation or configuration for ECS (e.g. how to set environment variables, memory/cpu requirements in task definition).
- The primary use case is analyzing logs from ECS tasks. For the MVP, we assume logs are accessible as files (e.g. exported from CloudWatch or mounted from an ECS task volume). In future versions, we plan to integrate directly with CloudWatch Logs, but for now the workflow might involve manually obtaining the log file from the container or CloudWatch.
- **Dual Interface in One Container:** The same container/binary should serve both interfaces:
- If run with a specific command or port, it can act as a web server for the UI (serving HTTP requests for upload and returning results).
- The CLI functionality should be accessible via the container as well (either by running the container with a command mode or by having the binary detect if it's called with CLI flags vs running as a server). This could be achieved by, for example, having an entrypoint that by default starts the web server, but if the Docker command is overridden (e.g. `docker run incidentsage incidentsage --file log.txt`), it executes the CLI analysis.
- **Resource Usage in Container:** The Docker image should contain minimal dependencies to keep size small. Base it on a lightweight base image (such as Python-slim or Alpine if using Python, or a Go binary in `scratch` if using Go, etc.). The LLM model file might be the largest component, but try to keep overall image size reasonable.

- **Configuration:** Provide options via environment variables or flags for any configurable parameters (for example, to select a different LLM model, or to adjust the log level for the Incident Sage application's own logs). For MVP, the defaults should just work, so this is secondary.
- **CloudWatch Integration (future):** A note in the documentation that CloudWatch log ingestion is not in MVP but is on the roadmap. For instance, future versions might have an AWS Lambda or integration that pulls logs from CloudWatch and feeds them into Incident Sage automatically.

5. Other Functional Requirements

Aside from the main features above, the product should meet these additional requirements to ensure quality and usability:

- **Lightweight & Fast:** The tool should be optimized for fast performance, especially in the CLI usage. Analyzing a reasonably sized log (e.g. a few thousand lines) should complete in a matter of seconds (aside from the LLM processing which might be the slowest step). The code should stream or incrementally process logs if possible, rather than loading extremely large files entirely into memory. Avoid heavy library bloat to keep the binary light.
- **Scalability (MVP scope):** Support analysis of log files up to ~100MB in size. This ensures it can handle substantial logs from a production incident. If a log is larger, the tool should handle it gracefully (possibly by chunking the log for the LLM or by informing the user the file is too large). For MVP, 100MB is a reasonable upper bound to test.
- **Graceful Handling of Malformed Data:** The application should not crash or hang on encountering malformed log entries or unexpected data. It should catch exceptions (e.g. JSON parse errors) and either skip those lines with a warning or attempt to extract whatever partial information is possible. This is crucial because real-world logs often have inconsistencies.
- **Clear Error Messaging:** All user-facing error or status messages should be clear and actionable. For example, if the log file cannot be read, the message might be "Error: Cannot open file `logs/app.log` – please check the path." If JSON parsing fails on some lines, perhaps "Warning: Skipped 10 malformed log lines." Avoid obscure stack traces or internal errors leaking to the user in normal operation.
- **Basic Unit Tests:** Even within a 4-day MVP, include a suite of basic tests for critical functions (e.g. JSON parsing function, incident detection logic, maybe a stubbed LLM summarization function). This ensures that future changes or refactoring can be done with confidence. Prioritize testing the log parsing and incident detection components. If the LLM call is hard to test, consider designing it with an interface that can be mocked (for example, have a mode where a dummy summary is returned for testing).
- **Documentation & README:** Provide a clear README.md (or equivalent documentation) in the repository. This should include:
- Setup instructions (how to build from source, if applicable, and how to run the CLI and Web UI).
- How to run the Docker container, with examples.
- Sample usage for the CLI, and a screenshot or description of the Web UI.
- Description of the local model usage and how to change the model (if applicable).
- Any prerequisites (for example, if using llama.cpp, mention that it will download a model on first run, etc.).
- For open-source release, include license information and contribution guidelines (even minimal).
- **Open-Source Code Quality:** As this is an open-source project, ensure the code is reasonably clean and maintainable given the time constraint. Use clear naming and structure. Include comments

especially in complex parts like prompt construction. This will help external contributors understand the project.

Non-functional Requirements

These are the constraints and performance targets the system should meet:

- **Performance:** The local LLM inference should be optimized to respond in under **10 seconds per analysis chunk** on average hardware. If a log is very large and must be processed in segments, each segment's summary should ideally return within this time. This ensures the tool feels responsive. Using quantized models and optimizing prompt size will be important to meet this goal.
- **Resource Utilization:** Incident Sage must run within a container limited to **2 vCPUs and 4 GB of RAM** (a typical small ECS task size). The chosen LLM model and runtime should fit in this memory (for example, a 4-bit quantized 7B model is roughly 4GB in size [6]). CPU-based inference is acceptable, but if the environment has a GPU it could optionally use it. The application should not consume significantly more memory than available; it should handle memory pressure (perhaps by processing logs in chunks rather than loading 100MB all at once).
- **Scalability & Throughput:** The tool is not meant for high-throughput concurrent requests in MVP. It will likely handle one log analysis at a time (which is fine for the use case of on-demand incident analysis). However, it should be able to handle multiple incidents within one log file. In a future version, if used as a service, it might need to queue requests; for now, this is out of scope.
- **Reliability:** The system should be stable during analysis. In case of any failure (e.g. LLM process crash or out-of-memory), it should fail gracefully (perhaps an error message to the user) rather than hanging indefinitely. Logging of the tool's own operations (at least in a debug mode) would help troubleshoot issues.
- **Compatibility:**
- The CLI should run on Linux (target environment for ECS/Docker). Running on macOS or Windows (for local use) is a bonus but not strictly required for MVP. Docker deployment will ensure it runs anywhere Docker runs.
- Log format assumptions: primarily JSON logs, but the code should be flexible enough to be adapted for other structured logs in the future (e.g. key-value pairs or plain text with regex patterns). In MVP, focus on JSON structure.
- **Security:** Since no external connections are made (no API keys needed and no outbound data), the main security considerations are limiting the web UI (which has no auth) to local or internal network usage. The Docker image should not run as root if possible (use a non-root user in the container) to mitigate risks. Also, ensure that uploading a log via the Web UI cannot lead to arbitrary code execution (i.e. treat the uploaded file as data, not as a script).

Out of Scope (MVP)

The following features or considerations are explicitly **not in scope for the 4-day MVP**, but may be addressed in future iterations:

- **Real-time Log Streaming:** The MVP will not handle continuously streaming logs or live monitoring. There is no mechanism to tail logs in real-time or constantly watch an application's output. Users must provide static log files (or dumps) for analysis. (Future idea: an agent mode that streams logs into the tool for continuous monitoring).

- **User Management & Authentication:** The Web UI in MVP has no login or multi-user support. It's assumed to be used in a secure environment (e.g. locally or within a VPN). Adding user accounts, roles, or authentication flows (OAuth, etc.) is out of scope for now.
- **External Log Store Integrations:** Direct integration with systems like Amazon CloudWatch Logs, Elasticsearch, Grafana Loki, Splunk, etc., is not included. The MVP deals only with logs provided to it (via file or copy-paste). In the future, a connector or plugin could fetch logs from CloudWatch or other stores using their APIs, but MVP users will manually retrieve and supply the logs.
- **Custom Detection Rules:** The incident detection logic in MVP is fixed (based on error levels and basic heuristics). We will not provide a user interface or config to define custom rules (e.g. user-specified regex patterns or thresholds for anomalies). Power-users cannot yet customize what constitutes an "incident" beyond the default implementation. Future versions might allow a configuration file or rules engine for this.
- **High-Availability or Scaling:** Running Incident Sage in a distributed, high-availability manner (multiple instances, load balancing, etc.) is not addressed. The focus is on a single-instance utility. In an enterprise setting, one might run it on demand or as a batch job per incident, rather than as a always-on service at this stage.

Deliverables

By the end of the MVP development, the following will be delivered:

- **Source Code Repository:** The full source code for Incident Sage, likely hosted on a platform like GitHub (open-sourced). It should contain the CLI/Server code, documentation, and configuration files. The code will have a suitable open-source license.
- **Compiled CLI Binary:** A built binary (or executable script) for the CLI tool. This may be included in the repository or provided as a release asset. It allows users to run `incidentsage` on their machines (assuming they have the model and dependencies).
- **Docker Image & Dockerfile:** A Dockerfile is provided in the repo, and an image may be published (e.g. on Docker Hub) for convenience. The Docker image encapsulates the application (both CLI and Web UI) and the local LLM model setup. Instructions to use the Docker image (pulling it or building it) will be in the README.
- **Web UI Interface:** The web application (served by the same binary) accessible via browser. This includes the HTML/JS/CSS assets needed. It doesn't have to be a separate deliverable per se (it's part of the source), but it should be functioning when the container or binary runs.
- **Sample Logs & Usage Examples:** A set of sample JSON log files for testing/demonstration. For instance, a small log file with a known error pattern can be provided (under a `samples/` directory) so users can try running the tool and see what the output looks like. The README should reference these and perhaps include example command outputs (or screenshots for the web UI).
- **Documentation:**
- A README as described above, covering installation, usage, and development notes.
- If appropriate, a short **User Guide** or Wiki entry explaining how to interpret the output, or how to customize the model (if that's an option).
- Inline code documentation for developers (comments and maybe a brief architecture note).
- **Basic Tests:** A small test suite (and instructions to run tests, e.g. `pytest` or similar if Python) to verify that key features work. This helps ensure the deliverables are verifiable.

- **Docker Deployment Instructions:** Documentation of how to deploy to ECS (since that's a target). Possibly an ECS task definition JSON example or `docker-compose` example for local. While we may not actually deploy it in the 4-day window, we provide guidance so that someone could do so.
- **Open Issues/Future Work:** Although not a "deliverable" in the traditional sense, the project should include a list of known limitations or TODOs (perhaps as GitHub issues or a section in the README) to clearly delineate what is left for future development beyond the MVP (e.g. CloudWatch integration, real-time streaming, etc. from Out of Scope). This sets expectations for users and contributors about where the project is heading.

Conclusion

Incident Sage's MVP will deliver a focused set of features enabling offline AI-driven incident analysis for log files. In just a few days of development, the goal is to prove out the concept: that a lightweight local tool can parse logs, identify incidents, and summarize them effectively using an on-device LLM. This PRD outlines the essential requirements to achieve that. By adhering to these requirements, we aim to build a useful tool for DevOps teams that can be extended with more integrations and capabilities in the future. The end result will be a Docker-packaged application that one can run in an environment like ECS or locally, point at a log file, and swiftly get an intelligible incident report – all without sending any data to the cloud.

With these specifications, the development team can proceed to implementation, testing, and delivery of **Incident Sage (MVP)** within the tight timeframe, focusing on accuracy, speed, and privacy as the primary values of the product.

---

[1] [2] [4] [5] LogWhisperer – AI-powered log summarizer that runs locally (no OpenAI keys, no cloud) : r/devops
https://www.reddit.com/r/devops/comments/1kfyv61/logwhisperer_aipowered_log_summarizer_that_runs/

[3] Incident Summaries using LLMs
https://karlstoney.com/incident-summaries-using-llms/

[6] Run Llama 2 Locally in 7 Lines! (Apple Silicon Mac)
https://lastmileai.dev/blog/run-llama-2-locally-in-7-lines-apple-silicon-mac