NVIDIA Assignment - Report

Yehonatan Ezra

GitHub Repository: `github.com/YehoanatnEzra/Nvidia_Assignment`

1 Introduction

In this document, I outline the design decisions, structure, and testing process I followed while implementing the log analyzer.In the following sections, I will briefly explain the project's structure, discuss key architectural decisions, describe the edge cases I tested, and highlight how each module contributes to the overall design.

Enhancements and Additions

Beyond the basic assignment requirements, this solution includes several thoughtful design improvements aimed at performance, flexibility, and clarity:

- **Multithreading:** The tool uses `ThreadPoolExecutor` to concurrently process both log files and filtering configurations. This significantly improves performance on multi-core systems, especially for large datasets.
- **Caching:** A `@cached_property` is used to store the result of the full analysis step, allowing repeated exports (e.g., console + JSON) without redundant computation.
- **JSON Export:** Users can decide to export the filtered results as a structured `.json`.
- **Compressed Logs Support:** The system supports both plain text (`.log`) and gzipped (`.log.gz`) files, enabling compatibility with production environments where logs are archived.
- **Smart CLI UX:** A dedicated `cli.py` wrapper provides user-friendly command-line parsing, validation, help messages, and interactive export prompts. Clear error messages and graceful failure paths are implemented.
- **Testing:** The project includes a complete `tests/` folder with individual test files for each core component:
  - `test_analyzer.py`
  - `test_config.py`
  - `test_entry.py`
  - `test_filter.py`

  Tests were written using `pytest`, and they verify both normal behavior and edge cases.

Object-Oriented Design

The project was carefully designed using object-oriented principles to ensure modularity, readability, and testability. Each class encapsulates a specific responsibility and exposes a clean interface.
Key OOP principles reflected in the implementation include:

- **Single Responsibility:** Each class is dedicated to a single concern - such as parsing log entries, filtering events, or managing the full analysis pipeline.
- **Encapsulation:** Internal data and logic are hidden behind well-defined class interfaces. For example, timestamp validation and parsing logic is encapsulated inside `LogEntry`.
- **Composability:** The system avoids deep inheritance trees and instead favors composition. The analyzer composes different building blocks (parser, filter, config loader) without tightly coupling them.
- **Open for Extension:** The logic for event filtering is fully data-driven, so new filters can be added simply by editing the configuration file - without modifying existing code.

This OOP architecture sets the foundation for the project's structure, as described in the next section.

[width=0.25]Photo2.png

Figure 1: Enter Caption

2 Project Structure

Top-Level Files

- `cli.py` - The main entry point for the application. It parses CLI arguments, validates paths, initializes the analyzer, runs the analysis, and optionally exports results to JSON .
- `messages.py` - Contains user-facing message constants (e.g., welcome message, export prompts).

log_analyzer/ Package

- `analyzer.py` – The central coordination module. It:
  - Loads and filters log files (supports `.log` and `.gz`).
  - Applies event filters using `ThreadPoolExecutor` for parallelism.
  - Uses `@cached_property` to avoid repeated computations.
  - Supports both console output and JSON export.
- `log_entry.py` – Defines the `LogEntry` class:
  - Parses individual log lines.
  - Validates timestamp format and range.
  - Encapsulates level, event type, and message fields.
- `event_config.py` – Loads and parses the event configuration file (e.g., `events.txt`):
  - Supports flags: `--count`, `--level`, `--pattern`.
  - Performs validation and builds a list of `EventConfig` objects.
- `event_filter.py` – Defines the `EventFilter` class:
  - Applies filters (event type, level, regex) to `LogEntry` instances.
  - Provides a clean `matches(entry)` interface.
- `error_messages.py` – Centralized error message definitions for formatting and consistency.

tests/ Folder

The project includes unit tests for every core module, written using `pytest`:

- `test_analyzer.py` – Tests full analysis workflow.
- `test_cli.py` – Tests CLI behavior, argument validation, and entry-point behavior.
- `test_config.py` – Verifies parsing of event rules and error handling in `event_config`.
- `test_entry.py` – Covers parsing, timezone assignment, and validation logic in `LogEntry`.
- `test_filter.py` – Tests correct filtering logic with different rule combinations.

Development Approach

The development followed a structured, test-driven methodology.
Each module was implemented incrementally and validated thoroughly using dedicated unit tests before proceeding to the next.

The process was as follows:

1. Developed the LogEntry class and its corresponding tests to ensure correct parsing and validation.
2. Built the `EventConfig` module and verified it using `test_config.py`.
3. Implemented the `EventFilter` logic and validated its correctness with multiple rule combinations.
4. Developed the core logic in `LogAnalyzer`, testing both parallel execution and filtering logic together.
5. Finally, created the `cli.py` wrapper to provide a clean and user-friendly interface.

This approach ensured confidence at each step of the development process, reduced debugging time, and enabled safe refactoring.

3 Design Considerations

This section provides a deeper explanation of the core architectural decisions made during the implementation of the assignment.

3.1 Multithreading and Parallel Processing

To improve performance, the analyzer dynamically sets the number of threads to match the number of available CPU cores using `os.cpu_count()`.
Specifically:

- Log files are parsed in parallel - each file is assigned to a separate thread, significantly speeding up file loading even with large volumes.
- Each event configuration is also processed in a separate thread, enabling rapid evaluation of multiple rules concurrently.

3.2 Result Caching with `cached_property`

The core analysis logic (loading and filtering all logs) is computationally expensive. To avoid repeating this step multiple times (e.g., once for printing and again for exporting to JSON), the result is memoized using Python's `@cached_property` decorator.

3.3 Object-Oriented Architecture

The design is intentionally object-oriented, not only for modularity and testability, but also to encourage future extensibility. Key patterns and benefits include:

- Clear separation of concerns: each class handles a single responsibility.
- Encapsulation of logic (e.g., timestamp validation lives inside `LogEntry`, not spread throughout the code).
- Data-driven extension: new event filters can be added without touching the codebase, just by updating the config file.
- Reusability and test isolation: each class can be tested independently with mock inputs.

4 Edge Cases Tested

This section outlines the key edge cases that were covered during testing to ensure system robustness and input validation. Unit tests were written using `pytest` and designed to catch malformed inputs, boundary behaviors, and unexpected configurations.

1. Timestamp Handling

- **Format Validation:** Ensures timestamps follow the ISO-8601 format (`YYYY-MM-DDTHH:MM:SS`).
- **Future Rejection:** Rejects any log entry that has a timestamp in the future relative to the system clock.
- **Age Sanity Check:** Prevents accepting entries older than a reasonable threshold (100 years by default).
- **Timezone Awareness:** Allows customization of the timezone using `ZoneInfo`. Defaults to `Asia/Jerusalem` if none is provided.

2. Log Parsing and Format Errors

- Missing fields in the log line (e.g., level, message, timestamp).
- Incorrect field order.
- Invalid timestamp components (e.g., 25:00 hours or 13th month).
- Mixed valid and invalid lines within the same file.

3. Configuration File Errors

- Unknown or unsupported flags (e.g., `--invalid`).
- Missing values after flags like `--level` or `--pattern`.
- Invalid or unparseable regex patterns (e.g., unclosed brackets).
- Ensured that flags work regardless of order.
- Handled blank lines and comments gracefully.

4. Event Filtering Logic

- Matching by `event_type` only.
- Matching by `event_type + level`.
- Matching by `event_type + regex pattern`.
- Matching by all three: `event_type + level + pattern`.
- Multiple filters for the same event type are handled independently.

5. File and CLI Behavior

- Skips files with invalid extensions (non-`.log` or `.gz`).
- Handles empty log folders without crashing.
- Filters entries properly using both `--from` and `--to` timestamps.
- Outputs matching entries or match counts as per `--count`.
- Successfully processes and combines compressed and uncompressed logs.