HTTPie Final Report

Dan Corcoran, Chris Shepard, Vinayaka Viswanatha

Project Overview

Our quality assurance project was performed on the HTTPie repository. HTTPie is a python based command line HTTP client with natural language syntax and colorized output. HTTPie is designed to sit in between the end user and the endpoint they are attempting to communicate with over HTTP and provide a simple interface for doing so. This lends itself to being very simple and straightforward to work with, increasing usability and performance. HTTPie supports various input and output formats such as files, JSON objects, and html forms. Its colorized and customizable output allows for developer flexibility and its built in session management, proxies, and authentication support ease of use. HTTPie works on both desktop and mobile making it a very popular tool for python developers with over 25 million downloads.

In this document, we will be outlining the various types of testing we performed on the HTTPie repository along with our major findings and what we learned from the overall process.

Testing Stages

Unit Testing Coverage Expansion

For Uniting Testing, our goal was to target paths of the codebase that had very low or no coverage. We added 15 new tests focusing mainly on utils.py, man_pages.py, plugins.py, and compat.py, which contain important helper logic, command-line integrations, and platform specific behaviour. Many of the execution paths in these files, especially those involving parsing logic and system interactions, were rarely tested under normal usage and had been overlooked by the existing tests. Our testing approach intentionally emphasized edge cases and exception handling to validate how the system behaves under less common but realistic conditions. This effort resulted in a 1% increase in overall project coverage, adding 38 newly added statements and improving confidence in the stability and correctness of the codebase.

Unit Testing Mocking and Stubbing

Next, for Unit testing with Mocking and Stubbing we focused on test behaviors that were hard to simulate normally - such as file I/O, SSL contexts, terminal outputs. 8 new unit tests were added using combination of MonkeyPatch, MagicMock, and patch to simulate file I/O operations, Rich console output, and SSL context behaviour. These techniques allowed us to isolate internal logic without relying on the actual filesystem, terminal, or network configuration, making the tests more reliable and easier to reason about. As a result, we added 23 newly covered statements, with rich_utils.py increasing from 0% to 72% coverage and writer.py improving from 83% to 93%, demonstrating the effectiveness of mocking in validating previously untested execution paths.

Mutation Testing

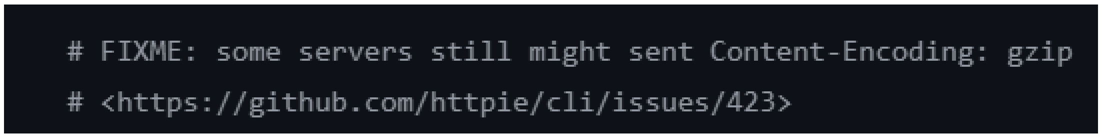
To evaluate the effectiveness of the existing and newly added unit tests, we conducted mutation testing using a tool called mutmut. Due to the size of the HTTPie codebase and the significant execution time required for a full-project mutation run (estimated at approximately 60 hours), the scope of mutation testing was intentionally restricted to the utils.py module. This module was selected because it contains critical parsing and helper logic that is widely used across the application. During this process, 3 additional test cases were added to specifically target surviving mutants related to cookie-splitting behaviour and URL parsing logic. As a result, the mutation score improved from 13 out of 33 mutants killed to 16 out of 33 mutants, highlighting logical execution paths that were not revealed by statement coverage alone and reinforcing the value of mutation testing as a complement to traditional unit testing.

Static Analysis

To perform our static analysis we utilized a tool called SonarLint which is a subset of SonarQube. SonarLint is a visual studio code extension and works directly inside of the text editor. SonarLint would scan each file and then as a warning report the issues it found, this allowed us to see the exact line numbers every issue was at and know exactly how to fix them.

Specifically, we ran the scan on every file in the source code, we did not run the scan on any start up scripts or documentation files as it felt unnecessary and doesn't directly affect the end user when they are using the program. Once doing so, we determined that the repository had a very high static code quality with very little issues reported.

The three main issues found were left over TODO/FIXME comments, large blocks of commented out code, and high cognitive complexity in a handful of functions. The leftover TODO/FIXME comments were the most concerning because there is no direct way to know if they have been addressed without going into a deep dive of the entire GitHub. Some of the comments linked to direct issues and PR tickets in the GitHub, but they were never addressed as seen in the image below.



```
# FIXME: some servers still might sent Content-Encoding: gzip
# <https://github.com/httpie/cli/issues/423>
```

As a team we decided to address the functions with high cognitive complexity because it provided a direct tangible benefit to the project. By breaking apart complex functions into smaller ones it promotes higher readability and modularity and it adheres to proper coding standards. In the end, we fixed three separate functions with cognitive complexities over 20 to_help_message(), parse_format_options(), and compute_new_headers() and reduced them to under 15.

Integration Testing

Integration testing was performed to validate the interaction between the core request-handling logic and session management in HTTPie. Three integration tests were added to verify that cookies are correctly persisted across sessions, expired cookies are not retrievable, and session or server cookies can be overridden as expected. These tests exercised the integration between core.py and sessions.py by invoking HTTPie commands through the CLI and observing real request–response behavior against a locally generated HTTP server. All tests passed successfully, and no fixes were required, indicating that the integration between session handling and the core controller logic is functioning correctly and meets expected behavior.

System Testing

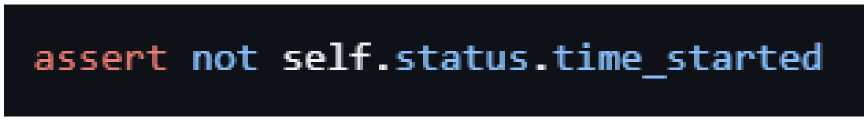
System testing was conducted using black-box testing techniques to validate HTTPie's behavior from an end-user perspective. Three system tests were added to verify the standard GET request workflow, proper handling of invalid or unreachable URLs, and end-to-end authentication using the HTTPie CLI. These tests interacted with the system exclusively through the command line without relying on internal modules, ensuring that core user-facing workflows behaved as expected. All tests passed successfully, with observed results matching expected outcomes, and no significant issues were identified. Overall, the results confirm that HTTPie functions reliably and correctly during typical usage scenarios.

Security Testing

We examined vulnerabilities inside of the /httpie directory. This does not include documentation, test cases, or some execution scripts found in other files. The /httpie folder holds the core logic of the application and is where all of the source code resides. All types of vulnerabilities were targeted. To analyze the program we used a tool called Bandit. This is a CLI vulnerability scanning tool which is able to statically analyze python source code to find potential issues. After the Bandit tool is run, a report is printed to the console detailing which vulnerabilities were found (low / medium / high) and how confident the tool is in that analysis (low / medium / high).

Upon doing so, we found ~30 CWE vulnerabilities across ~4 different categories all ranging from high, medium, and low. The three most prominent vulnerabilities were CWE 703 (Assert Used), CWE 400 (Uncontrolled Resource Use), and CWE 295 (Improper Certification Validation).

CWE 703 was the most concerning to us, this is because HTTPie assumes the endpoint has built in security measures and timeouts which implicitly solves CWE 400 and CWE 295.



```
assert not self.status.time_started
```

When running a python program, you can add in certain flags which ignore assert statements, so, when HTTPie is running in the background certain logic or code paths will never run which can certainly cause unintended logic. To fix this issue it is very simple, replace every assertion with a simple IF/ELSE statement. The core logic remains the same and the program is no longer vulnerable to CWE 703.

Performance Testing

Performance testing was conducted to evaluate HTTPie's behavior under sustained load, spike conditions, and increasing levels of concurrency. Tests included high sustained request volumes over several minutes, extreme request bursts over short durations, and scenarios involving a large number of simultaneous threaded requests. User connections were gradually scaled to several hundred over a short time window, as well as rapidly spiked to observe system response. These tests showed that CPU and memory usage could be easily driven upward, as HTTPie does not implement explicit resource clamping or throttling mechanisms. While this made it trivial to induce resource pressure, the behavior aligned with HTTPie's intended use as a lightweight development and testing tool rather than a load-testing framework.

Significant Issues

Major Finding #1

HTTPie does not have any sort of control structures pertaining to CPU or RAM utilization. Therefore, it's trivial to create tests that push memory and compute usage up to and beyond their maximums, going so far as to induce a system crash. This is exacerbated by data structure choices and the nature of Python garbage collection, meaning it's easy to start scaling memory requirements exponentially before stale memory is freed.

This issue, however, is not significant enough to warrant the large-scale refactoring that would need to take place. This is a development testing tool and is meant to verify small scale requests and endpoints. While stress, load, and spike testing are necessary for web development, there are other tools and frameworks that are more properly suited for the task. We have, in essence, attempted to test the pressure capacity of a fire engine with a garden hose. While technically possible, results will be catastrophic and are a reflection of the tool in use, not the quality of the software being examined.

Major Finding #2

While increasing unit test coverage improved confidence in the codebase, coverage metrics alone were not sufficient to assess the effectiveness of the test. Mutation testing revealed that several logical paths in utils.py were not adequately validated despite being marked as "covered", particularly in areas involving cookie parsing and URL handling. By introducing targeted test cases to kill surviving mutants, we demonstrated that mutation testing exposes weaknesses that statement coverage cannot detect, highlighting the importance of evaluating test quality in addition to test quantity.

Major Finding #3

A significant portion of HTTPie's codebase depends on system interactions such as file I/O, terminal output, subprocess execution, and SSL context creation, which are difficult to test using standard unit tests alone. Through extensive use of mocking and stubbing techniques, we were able to safely isolate and validate these behaviours without relying on the real runtime environment. This approach enabled meaningful testing of failure paths and edge conditions while keeping tests deterministic and fast, reinforcing the role of mocking as necessary strategy when testing CLI-based tools.

Improvements Made

Over the course of the project, several targeted improvements were made to enhance the testability, reliability and maintainability of the HTTPie codebase. Extensive unit and mocked tests were added across multiple modules, significantly increasing statement coverage and validating execution paths that had previously gone untested. These additions also contributed to measurable improvements in mutation testing results, demonstrating stronger test effectiveness beyond basic coverage metrics. In parallel, functions identified by SonarCube as having high complexity were refactored into smaller, more modular helper functions, improving code readability and ease of maintenance. Security analysis using Bandit further helped identify and document CWE-tagged issues, providing greater visibility into potential risks. Finally, performance and stress testing highlighted limitations in resource management under high load, which were documented as expected constraints given HTTPie's intended use as a development-focused CLI tool rather than a large scale load testing framework.

Quality Assessment

Both before and after this project the quality of HTTPie was high. The codebase was solid with high code coverage and very few issues. The tool itself worked when downloading it from pip and required very little oversight to do so. This high quality comes from its simplicity, HTTPie knows exactly what it needs to do and doesn't stray too far from the path. A simple codebase allows for higher levels of understanding and overall quality. Maintainability and testability remained largely unchanged over the course of the project since we didn't have to make any major changes to the code base.

We found static analysis and unit testing to be the most effective. Static analysis was incredibly easy to perform and provided us with many things that can easily be fixed along with insight into how to fix them. Unit testing is very industry standard and important, mocking and stubbing allowed us to greatly improve the overall code coverage metrics in many files.

There remain some risks and areas of improvement in the project. First of which, some functions with high cognitive complexity still remain. Although these functions are tested and perform as expected, they should still be broken down into more modular components to allow for reuse. Second of which, there still exists many untested exception handling conditions. Although the content inside of the functions performs correctly, the exception handling in the case something fails remains largely untested. This is a very simple process and seems to be implicit, but should still be tested nonetheless. Lastly, the overall project setup and documentation seems to be insufficient. As a team we struggled to get our systems set up to run the project from visual studio code. There were no separate instructions for windows vs mac and many steps could have been explained in more detail.

Lessons Learned

Lessons Learned #1

In order to have a successful project, development and testing need to occur at the same time, they need to go hand in hand. Testing as you go is always better than leaving it towards the end. By doing this, you mitigate the problem of accidentally changing code or introducing unintended consequences as development progresses, especially on a team of people. When you introduce testing you also introduce more metrics you can collect to both quantitatively and qualitatively prove if you are producing quality software which can be very important. To make this testing process easier, as a developer you should always be designing your code around testability. By doing so, you will feel that testing is part of the development process instead of an obligatory step thrown in at the end. Testing is important and can not be overlooked during the quality assurance process.

Lessons Learned #2

One key lesson from this project was the importance of effectively leveraging existing testing tools and libraries rather than attempting to build custom solutions. Modern testing ecosystems provide extensive support for unit testing, static analysis, and quality checks, making it possible to achieve strong test coverage and code quality without complex integrations. The project demonstrated that good testing does not require reinventing testing frameworks or introducing unnecessary complexity; instead, selecting appropriate tools and using them correctly can significantly reduce development effort. Overall, choosing the right framework for the right task simplifies the testing process and allows developers to focus more on validating behavior rather than managing tooling.

Lessons Learned #3

This project reinforced the value of using multiple testing techniques to gain a deeper understanding of system behavior. Unit testing helped uncover hidden behaviors and edge cases early in development, reducing the likelihood of unexpected failures later on. Mocking proved essential for isolating dependencies and safely exercising failure paths that would be difficult to trigger in a real environment. Security scanning further contributed by identifying issues that impact code safety and long-term maintainability. Finally, performance testing demonstrated that such tests reveal system and design limits rather than simply measuring execution speed, providing important insight into how the software behaves under stress.

Team Members and Roles

Chris Shepard -> Created the document template and filled in appropriate sections which corresponded to the sections of the presentation. During the project contributed by doing ⅓ of the work and documenting findings along the way to include in each report.

Dan Corcoran -> Filled in appropriate sections which corresponded to the sections of the presentation. During the project, contributed approximately one third of the work and documented findings as necessary. The project instructions effectively guaranteed this share of the workload and this team stuck to the instructions fairly closely.

Vinayaka Viswanatha -> Filled in the sections corresponding to the presentation content and contributed appropriately ⅓ rd of the overall project work.