TaleTuner

CS 436
Project Report

Github:
https://github.com/safwanyasin/bookAI_cloud_server
https://github.com/safwanyasin/bookAI_cloud

Syed Shahmyr Shah 28527
M Safwan Yasin 30037

1. Introduction:

TaleTuner is a mobile application that grants users access to a comprehensive library of books, and to enhance the user experience and user interactivity with the app on a personable level, the application grants users the ability to create an account of their own on the application, and then use these accounts to amplify their experience on the application. Utilizing their personal accounts users are able to use the search functionality to find a desired book, and once they find their target, they have access to the wishlist feature, granting them the means to save these books for future reference; at the same time the reading list functionality will keep track of every single book that an individual user has already read. The user may also choose to publish their own books using a generative AI tool available directly within the application itself; thus allowing them the opportunity to use a prompt to create the story that they desire and publish that book of theirs onto the application.

To ensure that the application continues to perform seamlessly and the user experience is as smooth as possible, we leveraged the technology made available through the Google Cloud Platform (GCP). This primary means of which being a combination of the Cloud Run features, for the sake of easily scalable and containerized applications, Cloud Storage, for the purpose of preserving and storing our data, and the API services, whose rule it was to handle the influx and outflux of data.

To research the performance of our application, and ensure that it does indeed operate at an adequate level, we stress tested our system using Locust, a robust and scalable tool that grants us the ability to simulate a great range of user loads and measure how exactly our system performs below, at and above, our estimated average user load thresholds. This was done to accurately find out whether our mobile application, TaleTuner, could not only perform as well as expected, but better as well.

2. Cloud Architecture:

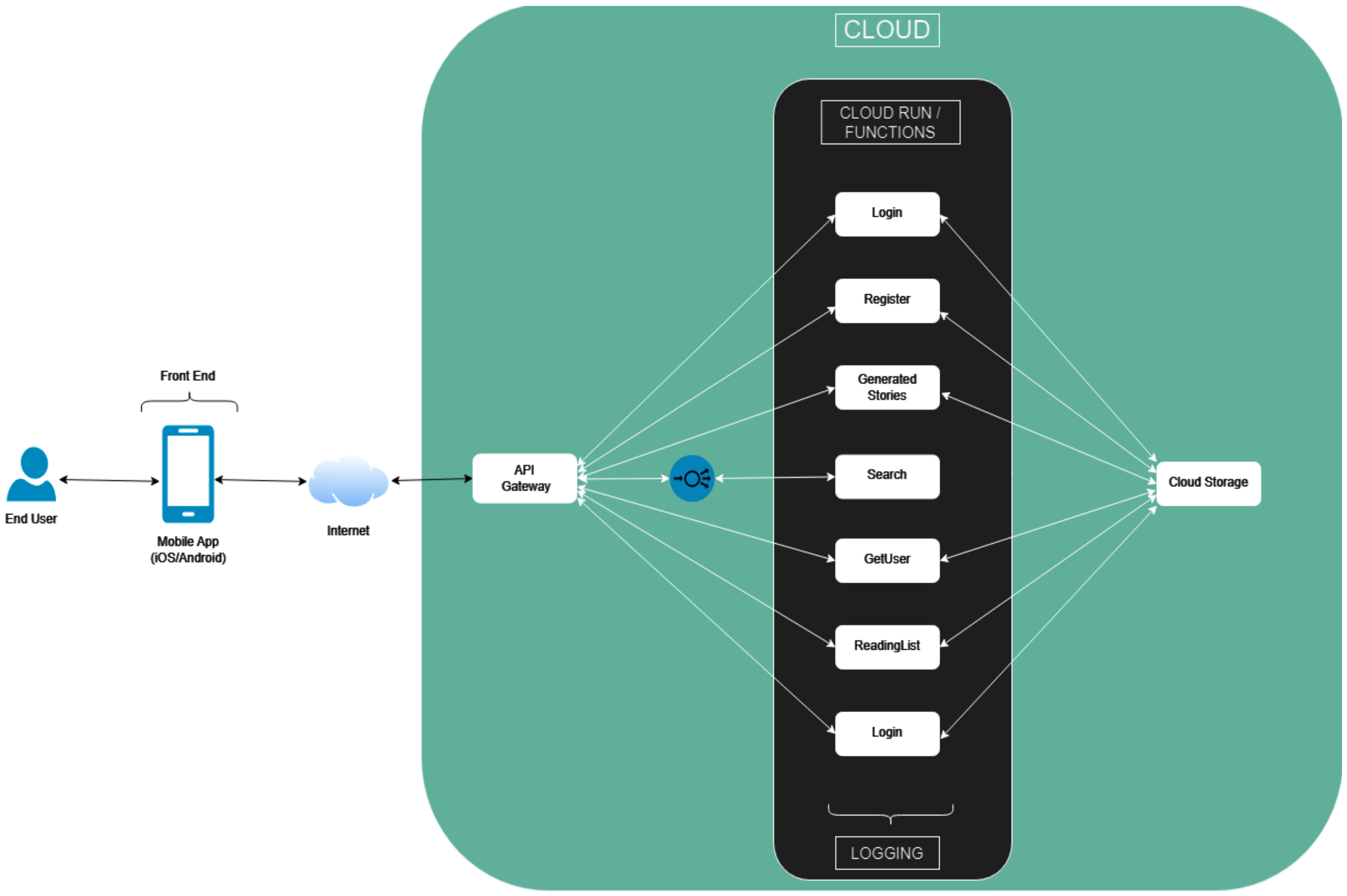

Figure 1.0: Cloud Architecture Diagram - Full Diagram also available on Github.

The user themselves will interact with the mobile application which is the front-end of our system, from there the application will connect to the internet and interact with our application back-end through API calls to and from the GCP Cloud Functions. What this means is that when the user wishes to login, register a new account, or use features such as the reading list, wish list or the generative AI functionality, then the app makes a call to the API which then goes to the respective Cloud Function with its request, and then that Cloud Function - which is also logging all its interactions - accesses the storage to gain the requested information and then sends that back to the application. The only Cloud Function that operates differently from the others, is the Search Function, which doesn't access the storage and instead is connected to through a load balancer, and makes API requests to an online book database, the reason for this is so that it can, in real-time, update the search results with the books as the user types.

3. Experiment Design

To systematically evaluate the performance of the TaleTuner application, we designed a series of experiments focusing on critical performance parameters. This section outlines the parameters considered, the performance metrics defined, the tools and configurations employed, and the steps taken to ensure the repeatability of the tests.

We identified key parameters that significantly influence the application's performance. The primary parameter was the number of concurrent users, varied from 15 to 400 to simulate different levels of demand. This range was chosen to encompass both typical usage scenarios and stress conditions. Additionally, each endpoint was subjected to a minimum of 10,000 requests to provide a robust dataset for analysis. The tested endpoints included login, register, search, reading list (get, add, delete), wishlist (get, add, delete), and generated stories (get, add, delete), all critical to the application's functionality.

To measure performance accurately, we defined several key metrics. Throughput, the number of requests handled per second, assessed the system's capacity to process concurrent requests. Latency, the time taken to process and respond to requests, was another crucial metric, with focus on average response time, median response time, and the 95th and 99th percentile response times. Error rates indicated the frequency of errors during requests, and resource utilization tracked CPU and memory usage to identify potential bottlenecks.

We used Locust for conducting the stress tests, chosen for its robustness and scalability. The configuration included user loads ranging from 15 to 400 concurrent users and each test involved at least 10,000 requests - as we had mentioned earlier - per endpoint to ensure statistical significance. A gradual ramp-up period was used to observe performance changes progressively and avoid sudden spikes. Tests were conducted over a sufficient duration to gather meaningful data and observe trends.

To ensure repeatability, we provided detailed instructions for setting up the test environment and executing the tests. This included comprehensive steps to configure the necessary hardware and software, including the GCP services used by the TaleTuner application. For test execution, we documented a step-by-step guide on running the tests using Locust, including configuration files and scripts, ensuring that others could replicate our experiments reliably and verify the results.

In summary, our experiment design for evaluating the performance of the TaleTuner application was thorough and systematic. By carefully selecting experiment parameters, defining detailed performance metrics, using robust tools and configurations, and ensuring repeatability, we obtained comprehensive and reliable data on the application's performance under various conditions. This approach allowed us to identify critical performance bottlenecks and areas for improvement, ensuring the TaleTuner application can effectively handle user demands.

4. Performance Evaluation

To evaluate the performance of the TaleTuner application, we conducted a comprehensive series of stress tests using Locust. This section delves into the results of these tests, examining key metrics such as throughput, latency, error rates, and resource utilization across all tested endpoints. Through this detailed analysis, we aim to uncover trends, identify bottlenecks, and propose optimizations to enhance the application's performance.

Our analysis began with the search endpoint, which exhibited significant latency issues under high load conditions. The average response time was recorded at 519.51 ms, with the 95th and 99th percentile response times spiking to 1100 ms and 1300 ms, respectively. These figures indicate that while the system handled a majority of requests relatively efficiently, a substantial portion experienced considerable delays. Furthermore, the endpoint encountered 170 errors, all "500 Internal Server Errors," highlighting a critical area for improvement. The high error rate and latency suggest inefficiencies in query handling and data retrieval processes that need to be optimized.

Similarly, the register endpoint demonstrated notable performance degradation as user load increased. With an average response time of 248.33 ms and significant spikes at the

95th (900 ms) and 99th percentiles (1100 ms), the endpoint struggled to maintain consistent performance under stress. The occurrence of 130 errors, also "500 Internal Server Errors," further underscores the necessity for improvements. These results indicate potential bottlenecks in validation and error handling mechanisms, which, if addressed, could significantly enhance the endpoint's resilience under high load.

In contrast, the login endpoint generally performed well, though it was not without its challenges. The average response time was 142.58 ms, with 95th and 99th percentile response times reaching 400 ms and 600 ms, respectively. While the error rate was relatively low at 60 errors, these figures suggest that there is still room for optimization, particularly in handling peak loads more efficiently.

The reading list endpoints, encompassing get, add, and delete operations, exhibited relatively good performance overall. The get reading list operation had an average response time of 112.63 ms, with spikes to 300 ms and 450 ms at the 95th and 99th percentiles, respectively. Despite encountering 50 errors, the endpoint maintained acceptable performance under load. The add and delete operations showed similar trends, with average response times of 142.88 ms and 90.82 ms, respectively. The error rates for these operations were low, indicating efficient handling of requests but still suggesting minor areas for optimization, especially during peak usage.

The wishlist endpoints displayed performance trends similar to the reading list endpoints. The get wishlist operation recorded an average response time of 112.63 ms, with the 95th and 99th percentile response times reaching 300 ms and 450 ms. The add operation, with an average response time of 142.88 ms, showed higher latency under increased load, while the delete operation maintained a lower average response time of 90.82 ms and did not encounter any errors. These results indicate generally good performance, but further refinements in backend logic and database interactions could improve efficiency during high demand periods.

Finally, the generated stories endpoints also demonstrated commendable performance, with the get operation averaging 112.63 ms and the add operation 142.88 ms. The delete operation, similar to the wishlist endpoints, maintained a low average response time of 90.82 ms with no errors. These endpoints handled the user load efficiently, suggesting that the system's architecture for managing generated content is robust, though continuous monitoring and periodic optimization could further enhance performance.

The stress tests revealed specific bottlenecks, particularly in the search and register endpoints, where high latency and error rates were prevalent. These issues point to inefficiencies in database queries and request handling that need to be addressed.

Implementing caching mechanisms for frequently accessed data, optimizing database queries, and enhancing error handling and validation processes are essential steps to improve performance. Additionally, scaling backend services to better manage high user loads will mitigate performance degradation.

In summary, while the TaleTuner application can handle up to 400 concurrent users, performance issues become apparent beyond this threshold. The detailed analysis highlights the need for targeted optimizations in specific endpoints to ensure consistent and reliable performance. Continuous monitoring and stress testing are crucial to maintaining high performance standards and ensuring the application meets user demands effectively.

5. Conclusion

The systematic performance evaluation of the TaleTuner application, conducted through a series of meticulously designed stress tests, has provided valuable insights into the application's ability to handle varying levels of user demand. By simulating loads ranging from 15 to 400 concurrent users and analyzing critical endpoints such as search, register, login, reading list, wishlist, and generated stories, we identified key performance metrics and trends that inform the current state and potential improvements for the system.

Our analysis revealed that while the application can efficiently handle up to 400 concurrent users, performance degradation becomes significant beyond this point. The search and register endpoints, in particular, exhibited high latency and error rates under stress, indicating bottlenecks in query handling, data retrieval, validation, and error handling processes. Optimizing these areas is essential to enhance the application's robustness and reliability.

In contrast, endpoints related to login, reading list, wishlist, and generated stories demonstrated relatively better performance, though they also showed areas for potential refinement. By implementing caching mechanisms, optimizing database queries, and scaling backend services, we can address these performance issues and ensure the system maintains high efficiency even under peak loads.

The detailed performance graphics, including throughput and response time percentiles, provided a clear visualization of how different parameters affect the system's performance. These visual aids, combined with comprehensive discussions of the results, facilitated the identification of performance bottlenecks and guided the development of effective remedies.

In conclusion, the TaleTuner application exhibits a solid foundation capable of supporting a substantial number of concurrent users. However, targeted optimizations are necessary to address identified bottlenecks and enhance overall performance. Continuous monitoring and periodic stress testing will be crucial in maintaining and improving the application’s performance standards, ensuring a seamless and satisfying user experience as the application scales and evolves.

Appendix

The performance evaluation graphs can all be found in the Github, in the test_reports folder, which can also be accessed through the following link:
https://github.com/safwanyasin/bookAI_cloud_server/tree/main/test_reports