Web Server Report

Suharta Rudra

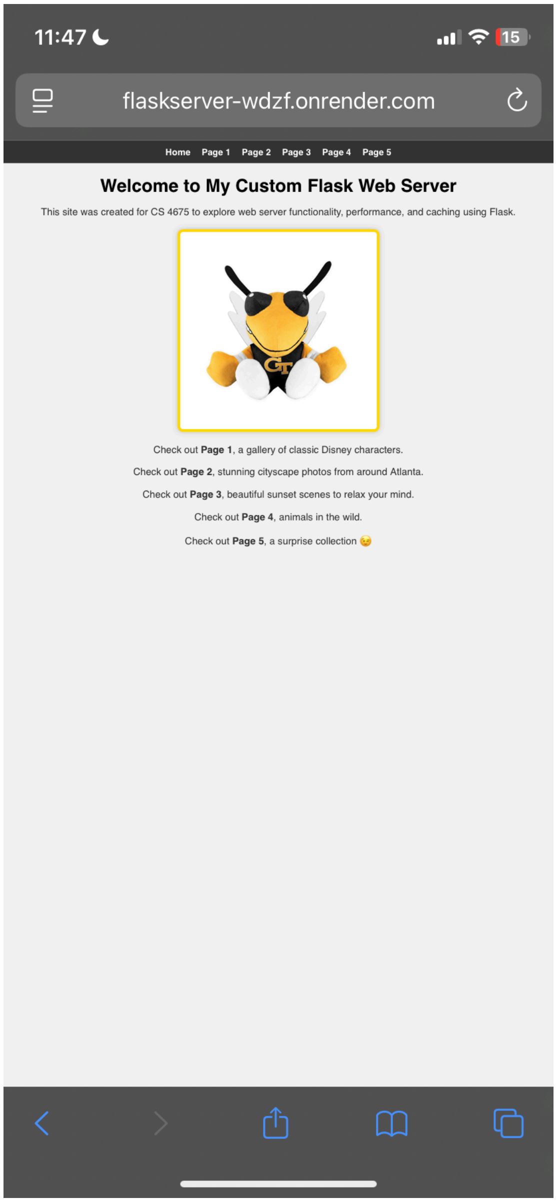
Github Repo: https://github.com/srudraDev/flaskserver
Website : https://flaskserver-wdzf.onrender.com/

Project Setup

For my web server project, I chose to use Python's Flask framework, as it's easy to use and powerful. The web server has 6 pages including the home page, which describes what the contents of my pages are in text. Each page displays high-definition images related to the heading of the page. Page 5 is the surprise page, and it includes a youtube thumbnail, a gif, and an image. The local server is configured to run on port 5000, but I decided to deploy my web server on the Render cloud platform, making it publicly accessible. When accessing on your mobile device, you can use the given website. If ran locally follow these steps:
1. Find your laptop's IP address as it acts as a server
2. Run the server using the command `flask run` (or `python app.py` to quickly see your laptop's IP address) on your laptop when inside the project directory
3. Go to http://<LAPTOP_IP_ADDR>:5000

Remote Client

First, I ran the server locally on my laptop and then accessed it using my iPhone that was connected to the same WiFi network to verify that the server is configured properly to listen externally and whether the design of my website was responsive and clean on a mobile device. Then I deployed the web server to Render, which gives me a public URL. I used the public URL for the screenshots below and all my performance and stress tests since real-world latency is built between the Render's servers and my laptop (that's how the Internet realistically works).



Welcome to My Custom Flask Web Server

This site was created for CS 4675 to explore web server functionality, performance, and caching using Flask.

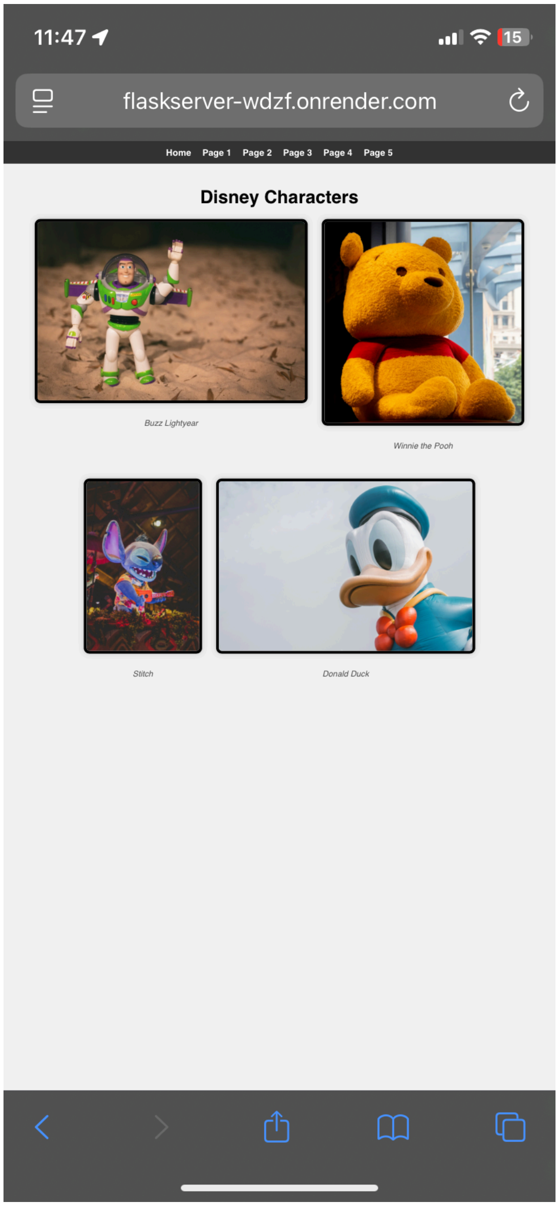
Check out **Page 1**, a gallery of classic Disney characters.

Check out **Page 2**, stunning cityscape photos from around Atlanta.

Check out **Page 3**, beautiful sunset scenes to relax your mind.

Check out **Page 4**, animals in the wild.

Check out **Page 5**, a surprise collection 😉



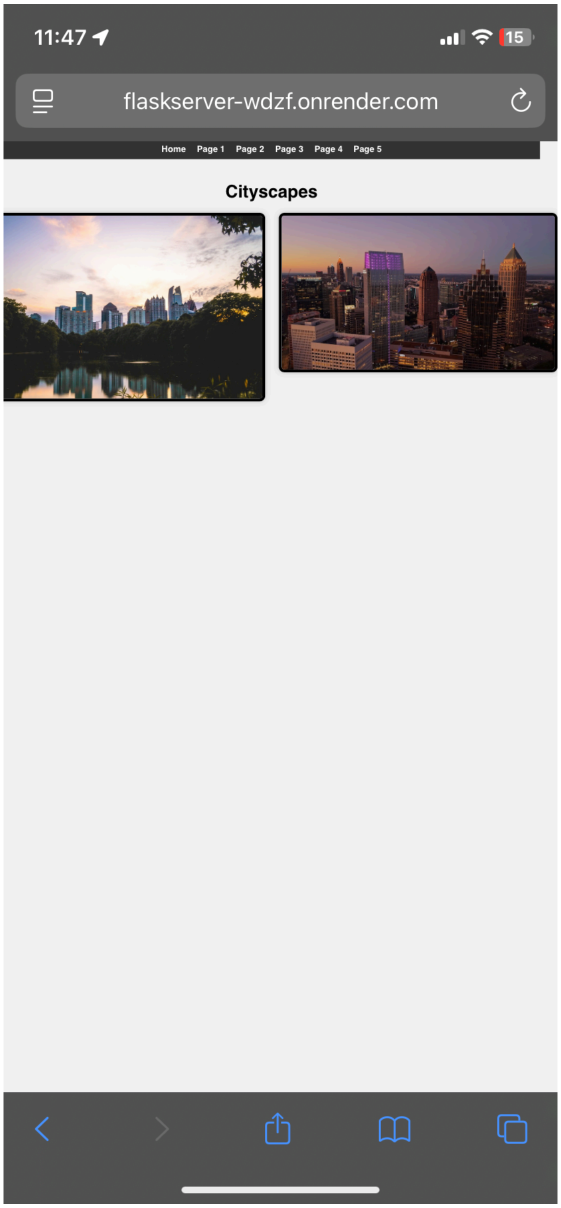
Disney Characters

Buzz Lightyear

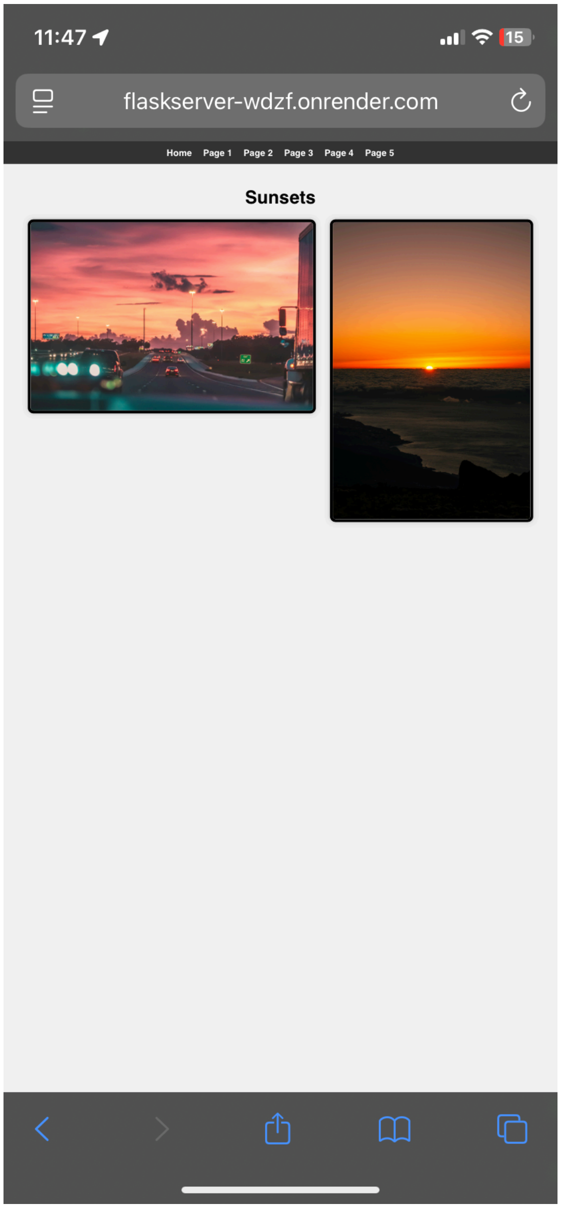
Winnie the Pooh

Stitch

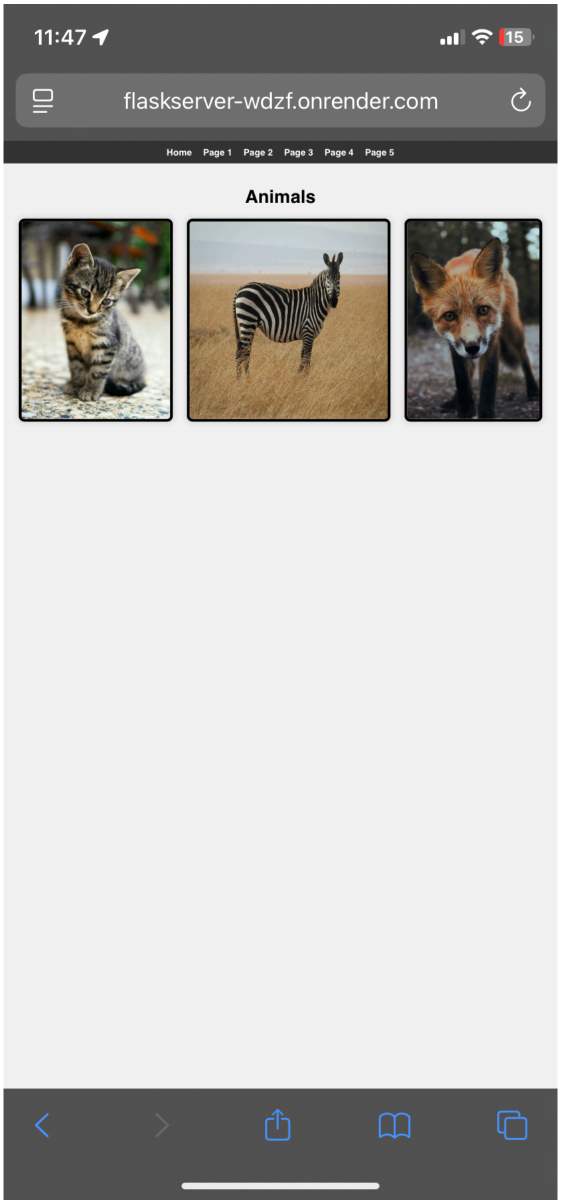
Donald Duck



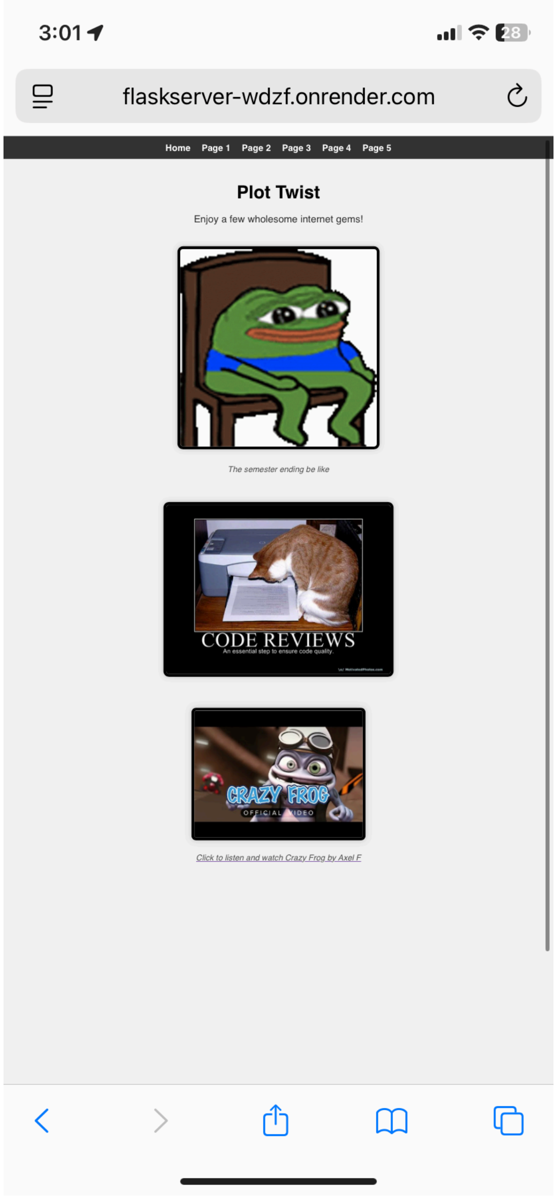
Cityscapes



Sunsets



Animals



Plot Twist

Enjoy a few wholesome internet gems!

The semester ending be like

Click to listen and watch Crazy Frog by Axel F

Performance Test

I used the ab tool (ApacheBench) which is commonly used for simple web servers to generate quick performance metrics. I recorded the following:

- Average requests per second (throughput)
- Average time per request (latency)
- Average data transfer rate
- 95th percentile response time - 95% of the requests completed in this time or less
- Maximum latency
- Failed requests

For each page, I ran the following command three time and calculated the average values:
`ab -n 1000 -c 10 https://flaskserver-wdzf.onrender.com/<page_path>`

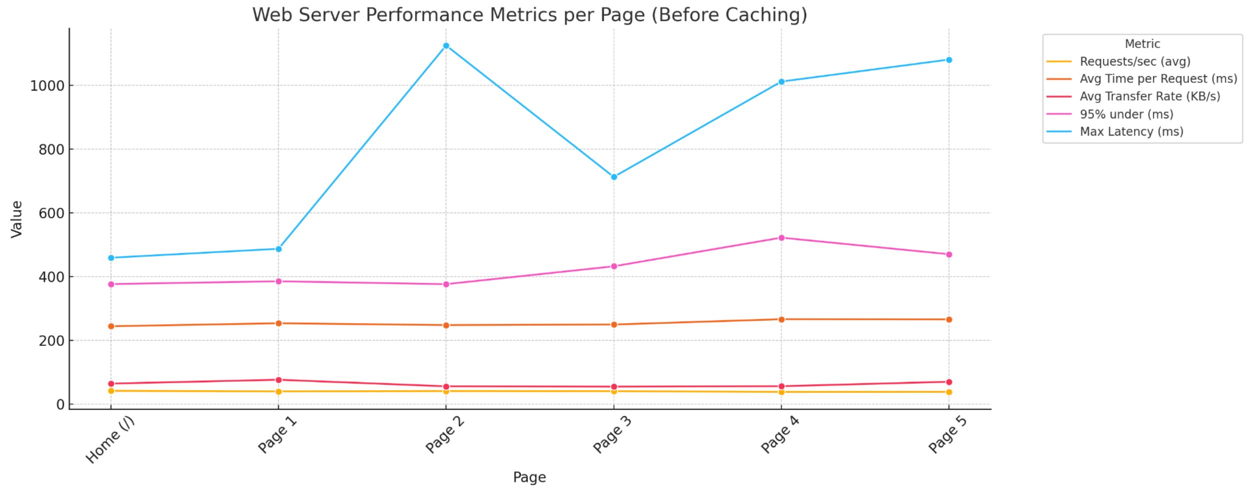
Where:

- -n 1000 sends 1000 total requests
- -c 10 simulates 10 concurrent users
- <page_path> is your designated page you want to test

Do note that the Render deployment configuration is using the free-tier environment. I made sure to spin up the instance before running any tests as inactivity can delay requests up to 50 seconds.

You can check out the raw results in the project repository under /results where ‘PerformanceBeforeCache.xlsx’ exists.

Here’s a chart generated through matplotlib that displays the same data.



<u>Discussion</u>

Pages with smaller HTML sizes had slightly better throughput, with up to ~41 requests/sec, whereas heavier pages like Page 4 and Page 5 handled fewer requests per second (around 37-38). Most pages had an average latency in the 240-265 ms range. Page 1 performed well with the highest transfer rate which makes sense due to its larger content size. Page 2,4 and 5 had the highest max latency, indicating small performance spikes under load or the fact that the second run for all three of them may be a host issue. But most importantly, no failed requests across all pages occurred during this regular test, indicating stable and reliable performance under modest concurrent load.

Stress Test

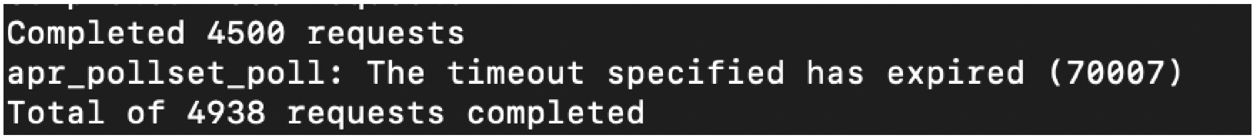
To evaluate the limits of my web server's performance, I conducted two stress tests using ApacheBench (ab) to simulate high traffic loads and observe how the server responds under increasing concurrency. I tested only /page1 and /page5 as those two pages had the most content as well as a high byte count. The scenarios I decided to test were 50 concurrent users and 3000 total requests and 150 concurrent users and 5000 total requests.

Here are the results for 50 concurrent users and 3000 total requests:

Page	Requests/sec	Time per Request (ms)	Transfer Rate (KB/s)	95% Under (ms)	Max Latency (ms)	Failed Requests
/page1	114.11	438.17	219.19	719	1839	0
/page5	70.84	705.77	130.19	692	15047	5

/page1 consistently outperformed /page5 in terms of throughput and latency, likely due to lighter processing or simpler content. /page5 had higher max latency and a few failed requests, meaning it's more resource intensive under load.

The results for 150 concurrent users and 5000 total requests were not producable as both failed to complete the test. ApacheBench printed progress in 500-request increments, but the test ended with the error: **apr_pollset_poll: The timeout specified has expired (70007)**



```
Completed 4500 requests
apr_pollset_poll: The timeout specified has expired (70007)
Total of 4938 requests completed
```

At this level of concurrency, the server became saturated, meaning it couldn't keep up with the request volume in a timely manner. Also, during the stress test, ApacheBench printed out this statement but continued on with the test:

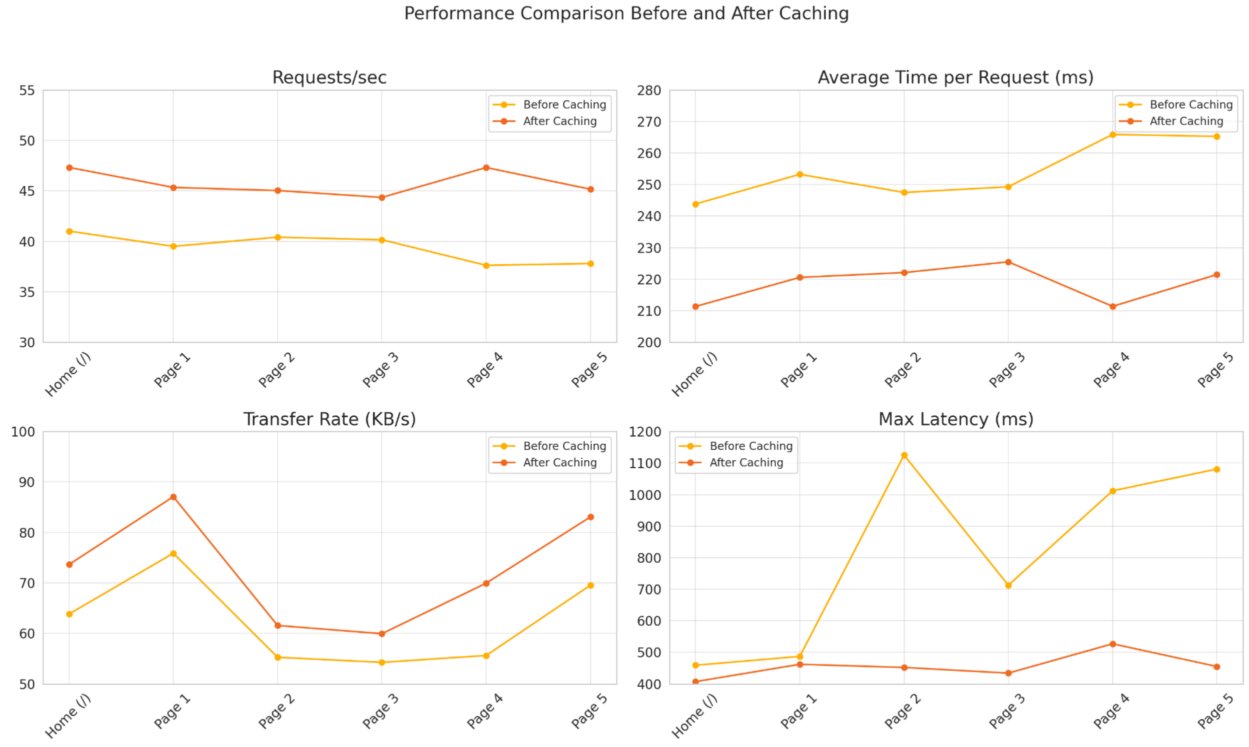
SSL read failed (1) - closing connection

This proves my interpretation as the server might have crashed or restarted due to server overload so some connections were dropped. Due to the deployment configuration being Render’s free tier, the server has limited capacity to scale beyond moderate concurrency levels. I decided to test and see if 100 concurrent users would allow me to see any results, but that also resulted in the same issue or the SSL read error. This demonstrates the limits of the current server setup and provides a foundation for future optimization such as upgrading deployment/server resources, or introducing load balancing to improve performance under high load. But the simplest solution would be to implement web-based caching which is the next section.

Caching Implementation

To enable server-side caching, I used Flask-Caching with SimpleCache, applying @cache.cached() to each route to cache the rendered HTML After deployment, I reran performance tests using ApacheBench except this time I did a cURL request to the page to establish caching as well as a dummy run. I observed a clear improvement in performance metrics from just the first runs, so here are the results:

There’s also a PerformanceAfterCache.xlsx file in the /results folder



And here are the results for the stress test:
50 concurrent users and 3000 total requests

Page	Requests/sec	Time per Request (ms)	Transfer Rate (KB/s)	95% Under (ms)	Max Latency (ms)	Failed Requests
/page1	119.79	417.4	230.1	602	1684	0
/page5	120.16	416.10	221.1	602	2504	0

150 concurrent users and 5000 total requests (worked this time):

Page	Requests/sec	Time per Request (ms)	Transfer Rate (KB/s)	95% Under (ms)	Max Latency (ms)	Failed Requests
/page1	158.98	943.51	305.39	1917	13776	0
/page5	165.2	907.99	303.94	1895	4597	0

Caching consistently reduced the average time per request and maximum latency, while also increasing throughput which is measured through requests/sec and transfer rate. All of the pages experienced improvement of 12-20% across metrics. Pages 1 and 5 saw the most improvement due to heavier content. Comparing the previous stress test table of 50 concurrent users and 3000 total requests, you can see the difference instantly. Before caching, our server couldn't handle 150 users for 5000 requests and now, it has avoided timeout and crashes and still no failed requests. This shows that just simple server-side caching with no client-side cache headers or CDNs for the static content can yield substantial performance benefits.

Summary

Through building and testing this toy web server with Flask and ApacheBench, I gained firsthand experience with how real-world web performance is measured, optimized, and affected by concurrency, caching, and content size. The theoretical concepts we have ingrained is now backed by real evidence and this project gave me a stronger appreciation for front-end performance under load, server configuration, and optimization. I've learned that:

1. **Latency under load is more sensitive to dynamic content generation than to static content size.**

While page size contributes to bandwidth usage, the real bottleneck under concurrent requests was the repeated rendering of templates on the server side. Even for relatively small pages, the server's performance degraded significantly when it was dynamically rendering the same template over and over again. Once I introduced server-side caching, this bottleneck was largely eliminated, thus implying that backend processing was the primary contributor to latency under load and not the file size.

2. **Server-side caching must be implemented first to improve performance**

Before even considering CDNs, load balancers, or cache headers, server-side caching should be mandatory for all applications that experience high traffic. Why make your users suffer with long load times when you can easily implement server-side caching? It dramatically increased requests per second by over 14% and latency dropped across all the pages in the web server. Imagine the results I would get if I had a better deployment configuration instead of a free-tier. Even if I increase from 50 to 150 users at a time like in the stress test, the server only processes the information once due to caching and gives them a Time-to-Live until the HTML contents expire.The error rate is 0% since there are no round trips between client and the server after visiting the site for the first time.

3. **Real-world web performance hinges on scalable design, not just code correctness.**

This project made it clear that building a functional Flask app is just the baby steps. Without accounting for production concerns like concurrency, caching, and client behavior, even a simple static site like blogsites and vanilla newspaper sites can become unresponsive under modest traffic. Running realistic performance and stress tests mirrored issues real websites face and it made me realize the importance of thinking like a systems engineer and not just a developer.