CS 188: Scalable Internet Services

PROFESSOR: John Rothfels

XCurrency

Final Report

Team Name: **XCurrency**

December 9, 2020

GitHub URL: https://github.com/scalableinternetservices/xcurrency

1. Introduction

XCurrency is an international currency exchange platform that matches exchange requests with other users to circumvent exorbitant rates that average 7%. The match algorithm only involves domestic money transfers. Here is an illustration of the algorithm.

Consider a scenario where Adam wants to transfer 1,000 USD from account A to account B. Brian wants to transfer 1314.52 CAD from account C to account D, and the requested rate is 1USD -> 1.31452 CAD, which is the live mid-market rate.

```
        1000 USD
A  ------------------------->  D
       1314.52 CAD
C  -------------------------->  B
```

1314.52 CAD is transferred from C to B and 1000 USD is transferred from A to D, which satisfies Adam and Brian's requests respectively.

2. Features

2.1 Log In/Sign Up

The website has an intuitive registration process that requires a name, email address, password, and password confirmation. The log in process only requires an email address and password.

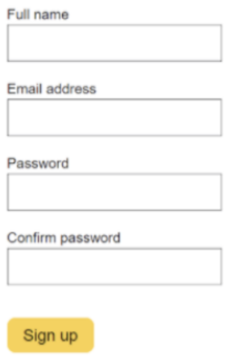


Figure 2.1.1: Registration Page Screenshot

2.2 Plaid Integration and Verification

The website integrates, verifies, and protects users' financial credentials with Plaid. The user can integrate with a myriad of banks through Plaid.

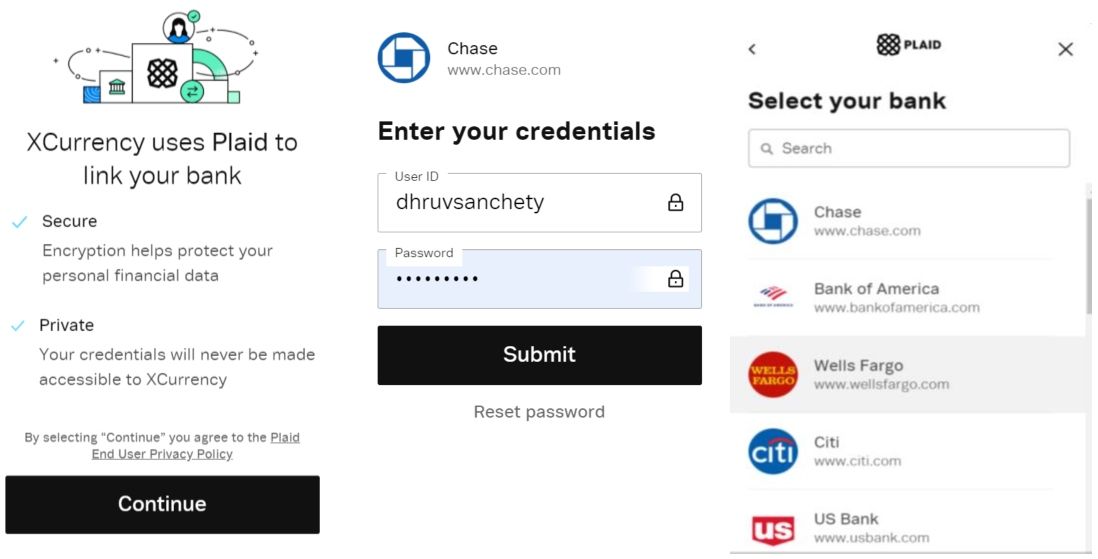
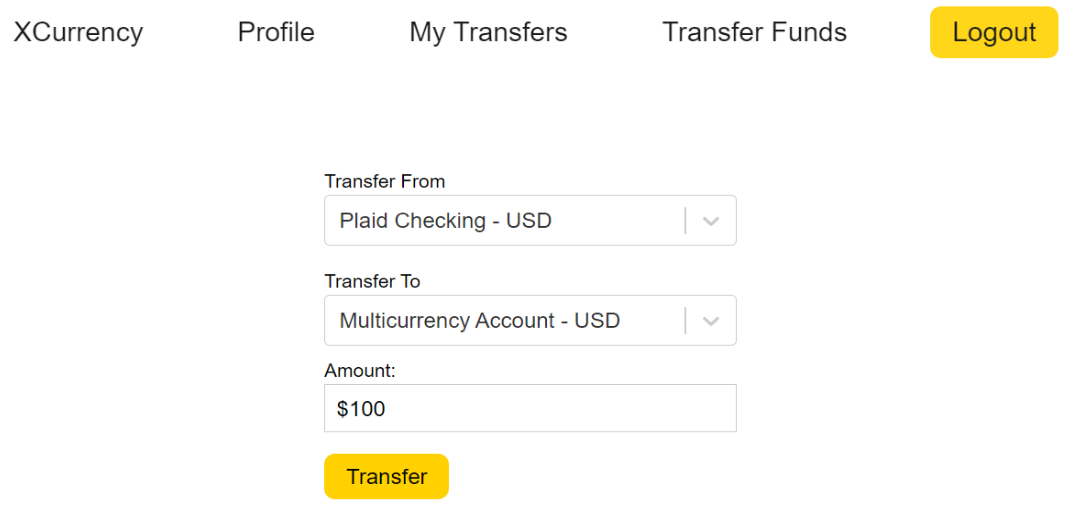


Figure: 2.2.1: Plaid Integration and Verification Screenshot

2.3 External and Internal Account Transfers

Users can perform money transfers to and from their external Plaid integrated bank accounts and their several internal multi-currency accounts.



Transfer From: Plaid Checking - USD

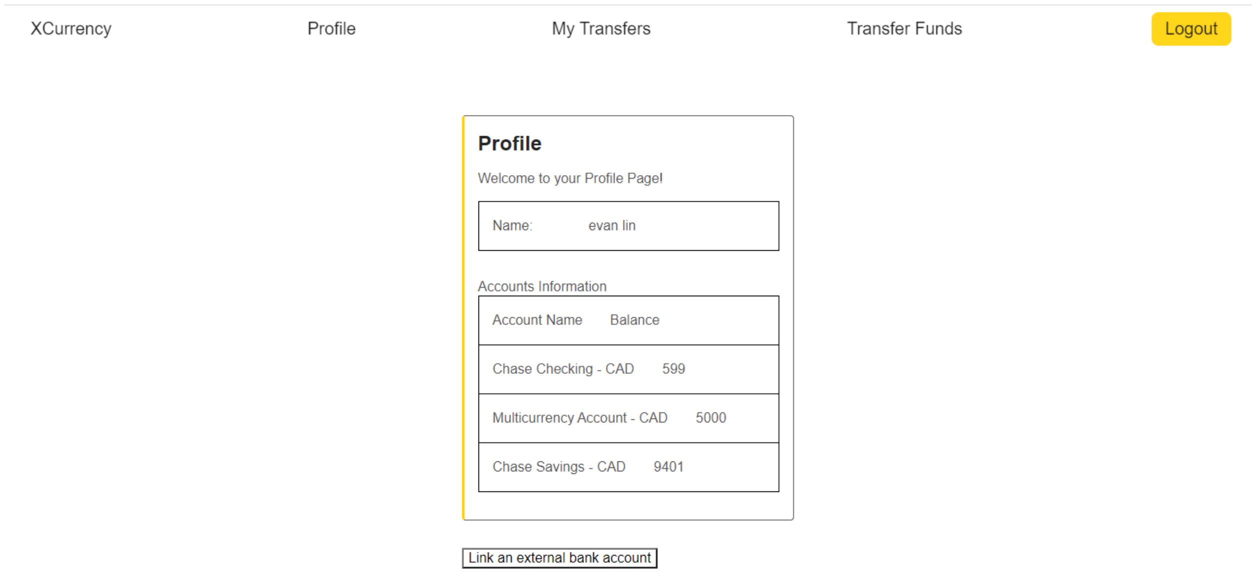
Transfer To: Multicurrency Account - USD

Amount: $100

Figure: 2.3.1: Transfer Funds from Plaid Checking to Multi-Currency Account.

2.4 Create, Maintain, and Visualize Accounts

Users can create and maintain multiple accounts in different currencies and can visualize their account details. They can utilize international interest rates to accumulate wealth.



Profile

Welcome to your Profile Page!

Name:	evan lin

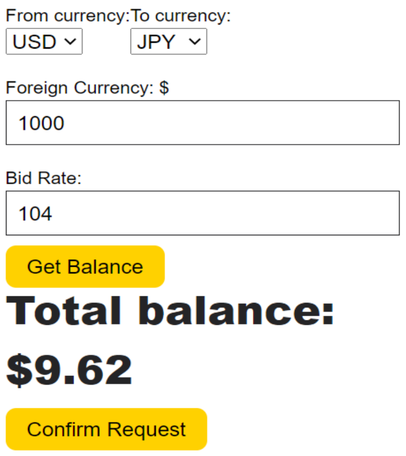
Accounts Information

Account Name	Balance
Chase Checking - CAD	599
Multicurrency Account - CAD	5000
Chase Savings - CAD	9401

Figure 2.4.1: User Profile Page and Account Details

2.5 Exchange Currencies

Users can request to trade their currency for another currency and specify their desired rate (bid rate), which will be matched with a request in the other direction. The website has 6 currency options that include the United States Dollar, Japanese Yen, Chinese Yuan, Indian Rupee, Canadian Dollar, and the Brazillian Real.



From currency: USD
To currency: JPY

Foreign Currency: $ 1000

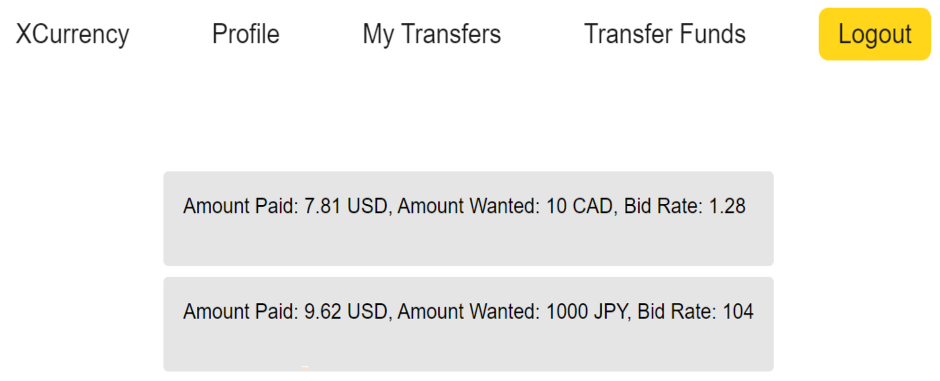
Bid Rate: 104

Total balance: $9.62

Figure 2.5.1: Trade Currency Request

2.6 Visit Current and Previous Requests

Users can view the details that pertain to their current and previous transfer requests.



Amount Paid: 7.81 USD, Amount Wanted: 10 CAD, Bid Rate: 1.28

Amount Paid: 9.62 USD, Amount Wanted: 1000 JPY, Bid Rate: 104

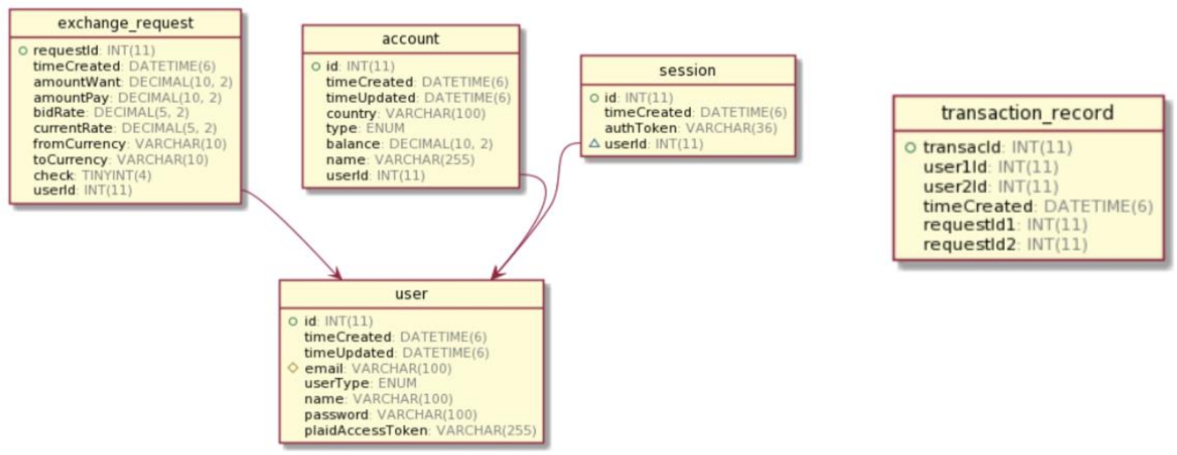
Figure 2.6.1: Money Transfer Requests History Page

3. Usage Scenarios

There are multiple use scenarios. Remittances or money sent to another country by international migrants are often the largest financial inflow to developing countries. Remittances exceeded $573 billion in 2019. Furthermore, tourists could maintain a multi-currency account that they could utilize in other countries. Expenditures for international tourism exceeded $1.575 trillion in 2019. In addition, traders who wish to speculate or predict currency fluctuations can use the website to trade currencies. Also, businesses that operate in other countries can use it to circumvent exorbitant rates. Moreover, individuals in countries that frequent inflation, currency devaluation, or currency depreciation can utilize it to protect their wealth. The USD ---> Argentine Peso rate increased by 700% and the USD -> Nigerian Naira rate increased by 200% since 2016. Individuals could also use the website to utilize international interest rates to accumulate wealth. The central bank interest rate minus the average inflation rate in Gambia is 13.26%. Last, international investors could use the service to reduce conversion expenditures and increase their returns.

4. Data Model

The data model includes the User entity. Each user has a name, email, password, a plaid access token, many accounts, and many exchange requests. Each account has a country, a balance that indicates the money in the account, and a user ID. Each exchange request has a string that indicates the original currency, a string that indicates the requested currency, a user, the current market rate, the bid rate, which is the requested rate at which the user wishes to transfer money, the amount in the original currency, and the amount in the requested currency. The data model also includes a transactions entity that has both users ID's and both requests' ID's. Figure 4.1 below illustrated how all of our classes and objects are defined and how they interact with one another.



<u>Figure 4.1:</u> Unified Modeling Language (UML) entity relationship diagram for our application.

5. Optimizations

5.1 Optimizing Server-Side Rendering

<u>Baseline Performance</u>

After all the features have been implemented, we began our optimization efforts by measuring the baseline performance of our application. To determine the queries per second (QPS) before performance degradation, we ran a K6 load test that used a constant arrival rate executor to simulate a fixed number of iterations per script execution. We then ran increasingly more intense K6 load tests and plotted the results in a P95 latency vs. QPS line graph to determine the QPS at the point of rapidly declining performance. The QPS for each load test was determined by multiplying the number of script iterations per second (a K6 parameter) with the number of queries in the specific script. This methodology is consistent across all each optimization's baseline performance analysis. Optimizations 5.1 to 5.6 were load tested on a quad-core Intel i5-6600 Desktop CPU using the Windows operating system and 16 GB of RAM. Load testing and Optimization at 5.7 and 5.9 were performed on 2.6GHz 6-core Intel Core i7, Turbo Boost up to 4.5GHz, with 12MB shared L3 cache.

For the initial baseline performance load testing script, we tested the landing page by making a single GET request to /app/index since we were already aware of the server-side configuration issue from lecture.

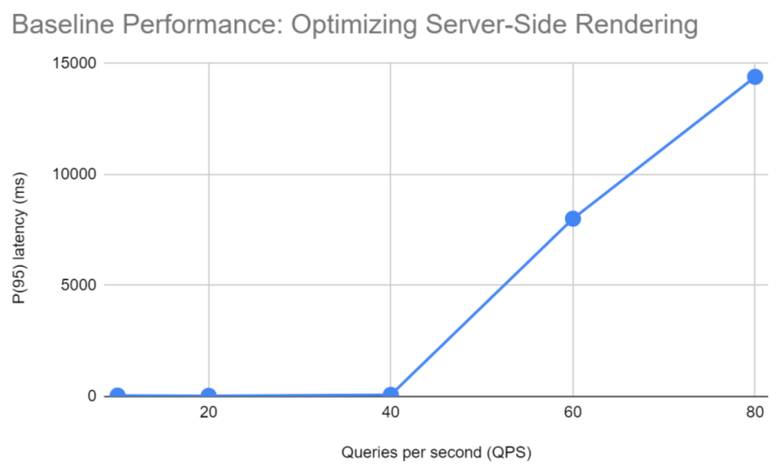
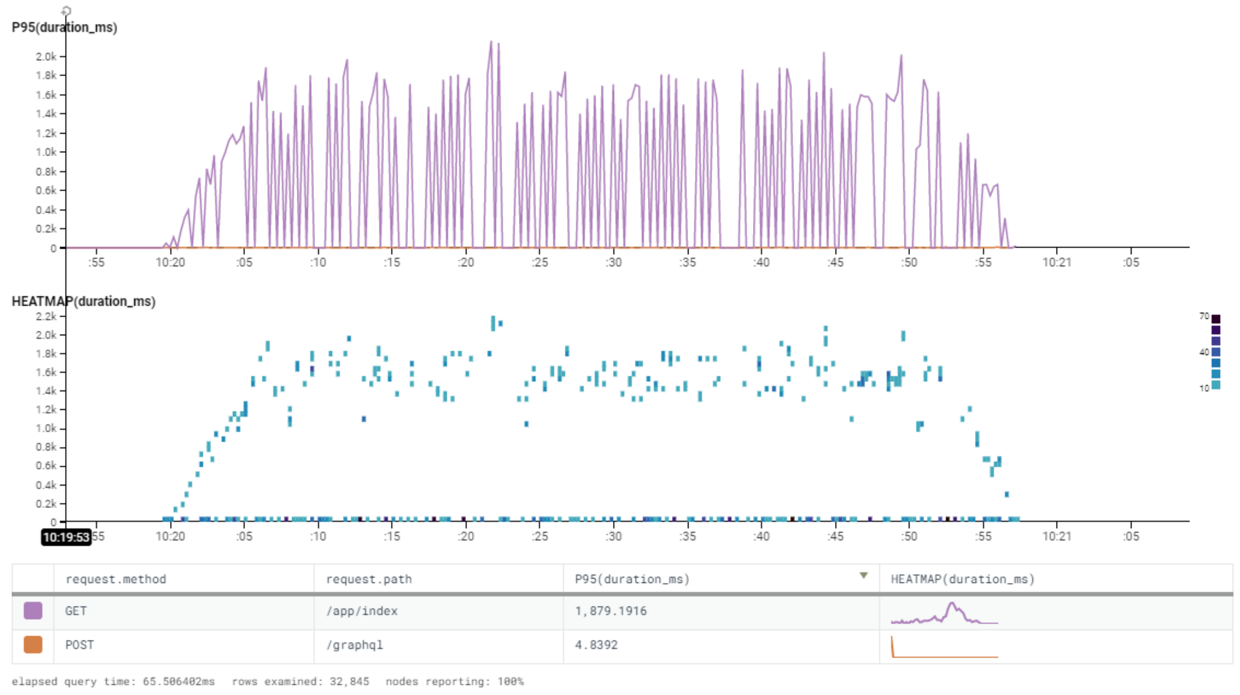


Figure 5.1.1: Baseline performance of our application before any optimizations.

From Figure 5.1.1, we can see that our application's performance drastically decreases when the QPS load reaches around 40 QPS, which is much lower than expected from a simple test scenario that makes one GraphQL request to fetch the user context for the landing page.

To gain more insight into the source of the issue, we ran a K6 load test using the ramping arrival rate executor to simulate a maximum load of 200 concurrent users by quickly ramping up to 200 concurrent users and ramping down to 0. We used this type of load script as our primary approach to diagnose potential optimizations to mitigate the effects of our system's performance on the load tests. As explained in the K6 documentation, the ramping arrival rate executor will consistently add load to our system regardless of whether our application server's response times decrease under increased load pressure. This decouples the effects of our server's response time from the test's intended amount of load.

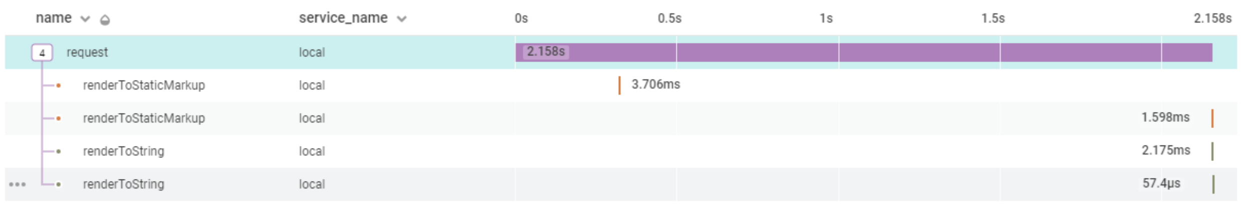


	request.method	request.path	P95(duration_ms)	HEATMAP(duration_ms)
🟪	GET	/app/index	1,879.1916	
🟧	POST	/graphql	4.8392	

```
elapsed query time: 65.506402ms   rows examined: 32,845   nodes reporting: 100%
```

Figure 5.1.2: Honeycomb P95 duration and heatmap distributions.

From Figure 5.1.2, we see that the majority of the latency issues are due to handling the request for /app/index, rather than the GraphQL query. The heatmap distribution for the GraphQL endpoint confirms this finding since all GraphQL requests receive a response in less than 20 ms.



name	service_name	duration
request	local	2.158s
renderToStaticMarkup	local	3.706ms
renderToStaticMarkup	local	1.598ms
renderToString	local	2.175ms
renderToString	local	57.4μs

Figure 5.1.3: Trace of one of the worst performing /app/index responses.

From the trace in Figure 5.1.3, we see that the call to renderToString very trivially contributes to the latency. The possible problematic statement in our code should be between the initial call to renderToStaticMarkup and renderToString, where there is a significant gap in time shown in the trace.

Hypothesis

In lecture, we discussed that this is an issue with our configuration of server-side rendering, not the inherent nature of server-side rendering itself. Rather than make a local call directly in the GraphQL layer, our server pays the additional overhead of starting up a HTTP connection with itself to call its own GraphQL endpoint each time.

We hypothesize that the proposed solution is to swap out the Apollo client's HttpLink with a SchemaLink that corresponds to the GraphQL layer. As this is the same optimization mentioned in lecture, this should solve the problem.

Implementation of Proposed Solution

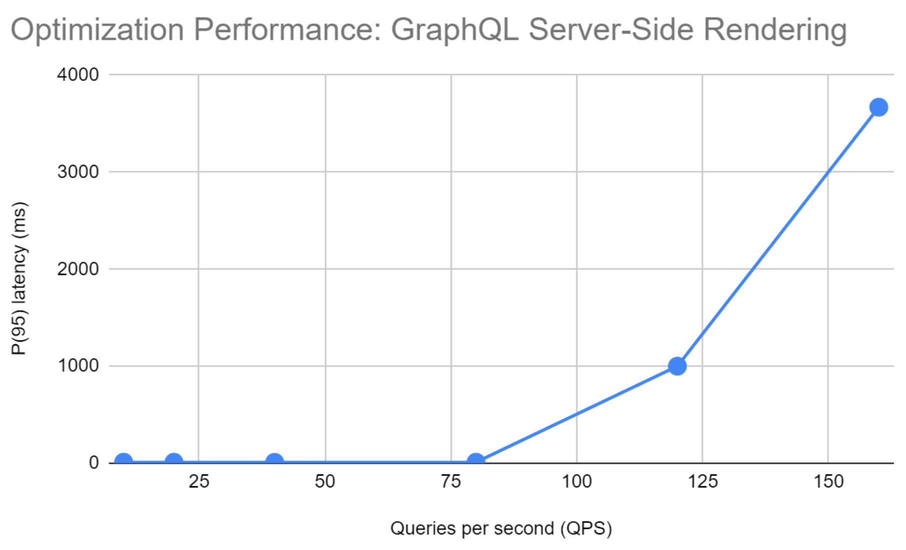
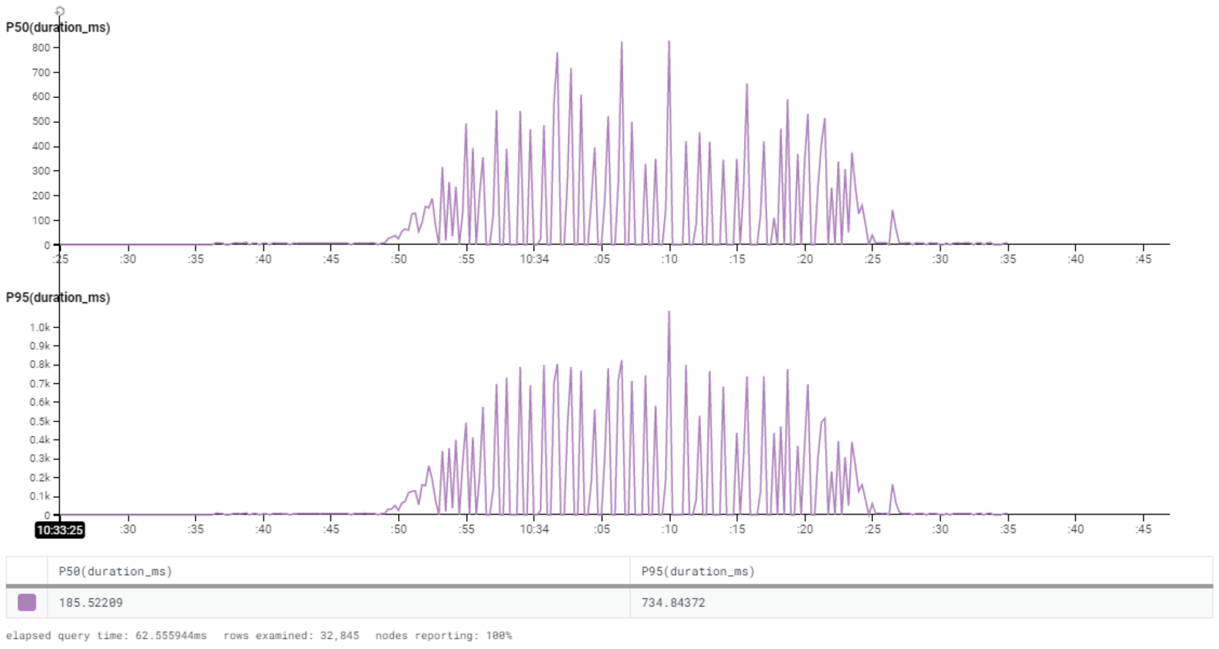


Figure 5.1.4: Max QPS Performance after implementing the server-side rendering optimization

From Figure 5.1.4, we see that the new maximum QPS before the server experiences a sharp increase in P(95) latency is around 120, a 300% increase in QPS. Still, this is actually much less performant than what we expected for a simple GET request. We suspect it may possibly be due to the limitations of using a constant arrival rate executor in K6 load testing, which requires a large amount of virtual users to be preallocated before the test runs and that may have reduced server performance.



	P50(duration_ms)	P95(duration_ms)
🟪	185.52209	734.84372

```
elapsed query time: 62.555944ms   rows examined: 32,845   nodes reporting: 100%
```

Figure 5.1.5: Post-optimization Honeycomb P99, P95, and P50 request duration.

From Figure 5.1.5, we see that there is an approximate 60.1% reduction in P95 latency, from nearly 2 seconds to 734 ms, when making a request to the landing page. Although a delay of 734 ms is still perceivable by the user, it is not enough to break a users' flow of concentration. In addition, we see that the median delay is only 185 ms, revealing that the majority of our users will fail to notice any delay.

5.2 Optimizing Bcrypt Library Selection Choice

Baseline Performance

Our first non-trivial script creates a new XCurrency account and authenticates to the application using the newly created account. This required two backend endpoint calls: a POST request to the /auth/signup backend endpoint and a POST request to /auth/login. We ran a baseline performance load test, as explained in more detail in Section 5.1, and produced the following results.

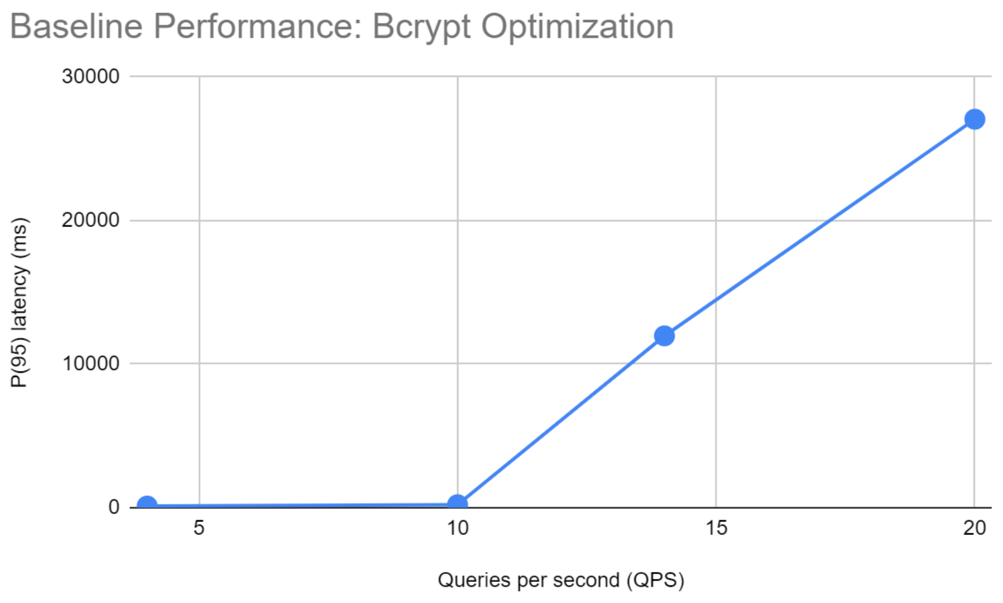
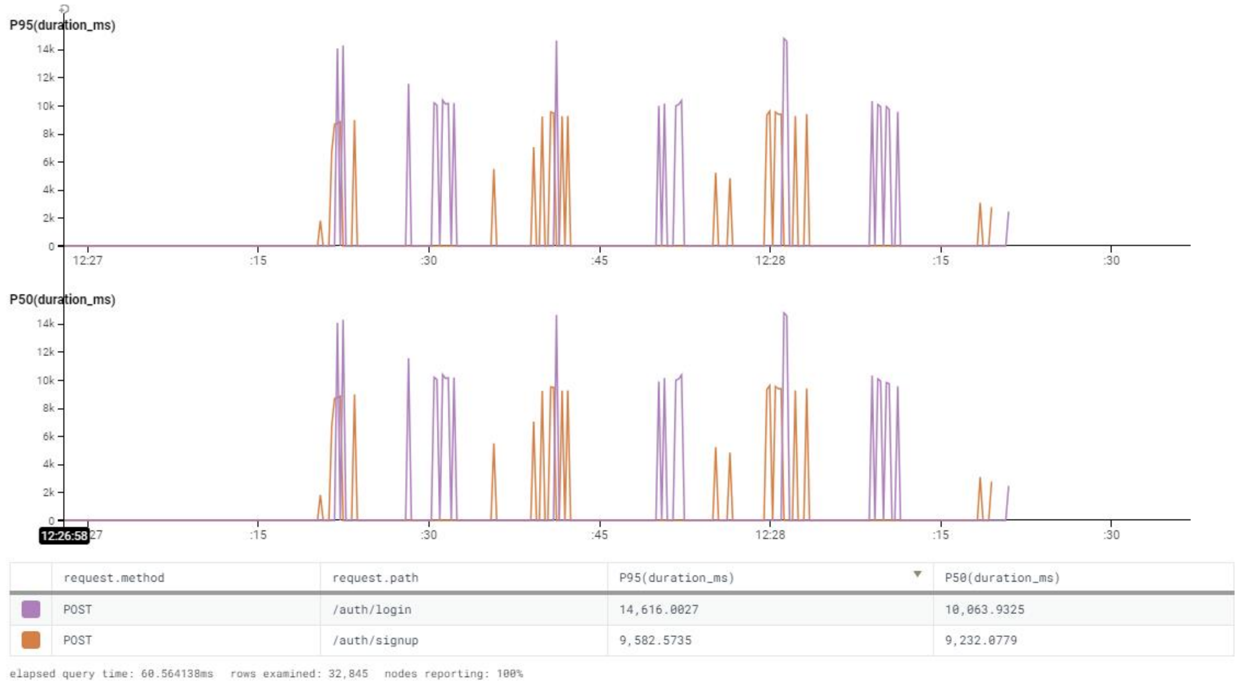


Figure 5.2.1: QPS baseline performance for login and sign-up

From Figure 5.2.1, we see that performance drastically decreases with a load of only 10 QPS. This was a surprising result because both endpoints did not appear to make very many database queries, and at first glance, not computationally expensive.

To gain more insight into the problem, we ran the same type of K6 ramping arrival rate executor load test using a maximum load of 200 concurrent users (as explained in Section 5.1) and produced the following Honeycomb P95 latency results for login and signup.



	request.method	request.path	P95(duration_ms)	P50(duration_ms)
🟪	POST	/auth/login	14,616.0027	10,063.9325
🟧	POST	/auth/signup	9,582.5735	9,232.0779

elapsed query time: 60.564138ms rows examined: 32,845 nodes reporting: 100%

Figure 5.2.2: Pre-optimization Honeycomb P95 latency

From Figure 5.2.2, we see that P95 values for each endpoint are unreasonably high for our users, requiring over a 14.6 second wait for logging in and a 9.5 second wait for signing up for 5% of our users.

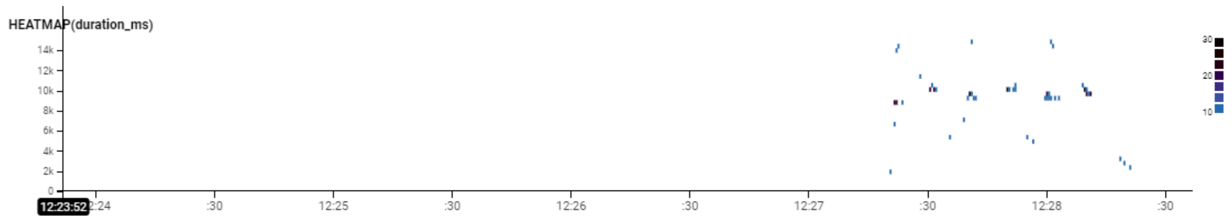
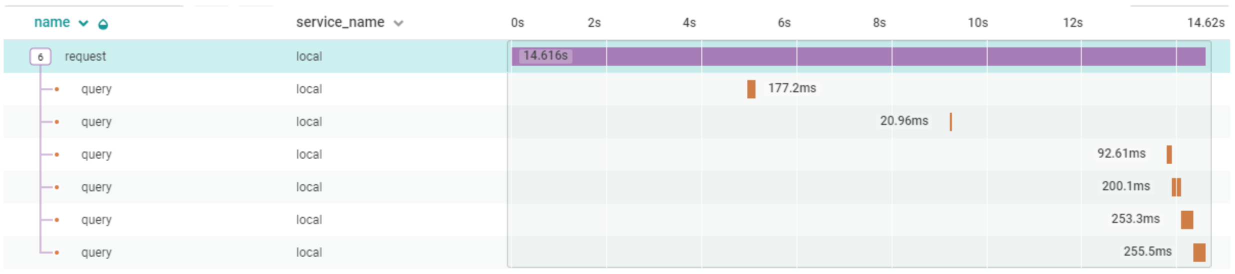


Figure 5.2.3: Pre-optimization Honeycomb heatmap latency distributions

From Figure 5.2.3, we see that the worst case wait times are distributed quite evenly, indicating that the issue appears for not just a small portion of users under load, but rather, most users seem to experience the issue. This is further reinforced in the P50 values shown in Figure 5.2.2, where we see that half of all users have to wait 9.2 seconds and 10.1 seconds to signup and login, respectively. To diagnose the issue, we examined the following trace.



name	service_name	duration
6 request	local	14.616s
query	local	177.2ms
query	local	20.96ms
query	local	92.61ms
query	local	200.1ms
query	local	253.3ms
query	local	255.5ms

Figure 5.2.4: Pre-optimization trace of one of the worser performing login requests

The first observation we made was that there was a consistent time gap between the first two database queries across all traces of /auth/login. This narrowed down the scope of potentially unoptimized lines of code down to a call to that used the bcrypt library to compare the user's hashed password in the database with the provided password in the login form. This led us to research more about the bcrypt hashing algorithm, which we had initially plugged into our code without thought to avoid storing our passwords in plaintext. Bcrypt is a computationally expensive algorithm designed to prevent brute-force attacks against hashed passwords by slowing down the rate at which hackers can hash potential passwords. At first, this seemed like an unavoidable tradeoff between security and performance, until we recalled the advice given in lecture to be distrustful of external libraries.

Hypothesis

We soon realized that we were using bcryptjs, a library that used purely JavaScript to compute hashed passwords and intended for browser use only. An alternative library, called bcrypt, appeared more performant because it ran on native C++ code. Theoretically, using C++ would allow the computationally expensive algorithm to run on a background thread and prevent the computation from blocking the NodeJS event loop for increased throughput.

We also noticed from Figure 5.2.4 the simple SQL queries to determine whether an entry existed in the User table appeared to take much longer than expected, about 177 ms in the trace. We ran an EXPLAIN query to determine the source of inefficiency, and discovered that the 'type' returned was 'index,' indicating that a full table scan was used when we could have scanned using an email index on the User table.

Our proposed solution was to switch over from the bcryptjs library to the bcrypt library and optimize our SQL query in the User table by adding the email property as an index.

Implementation of Proposed Solution

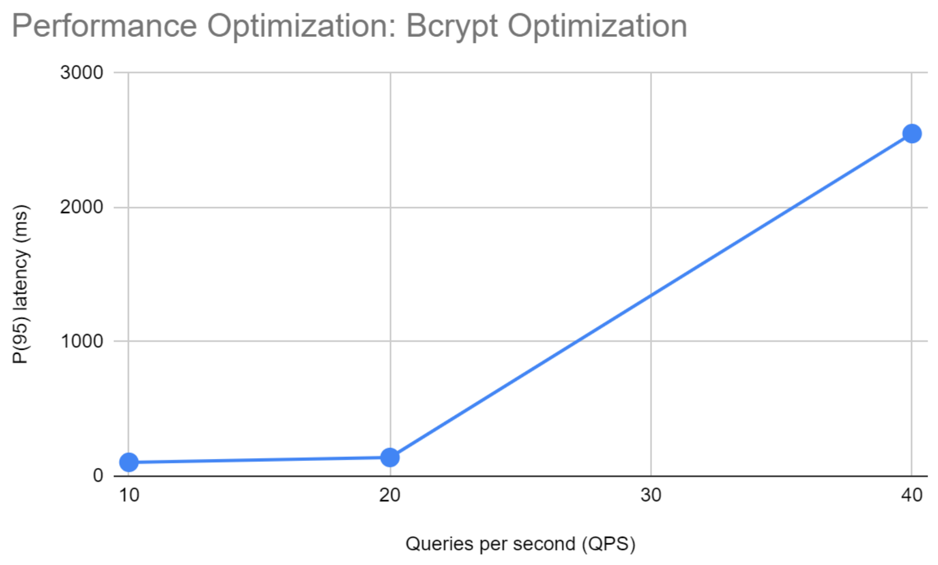
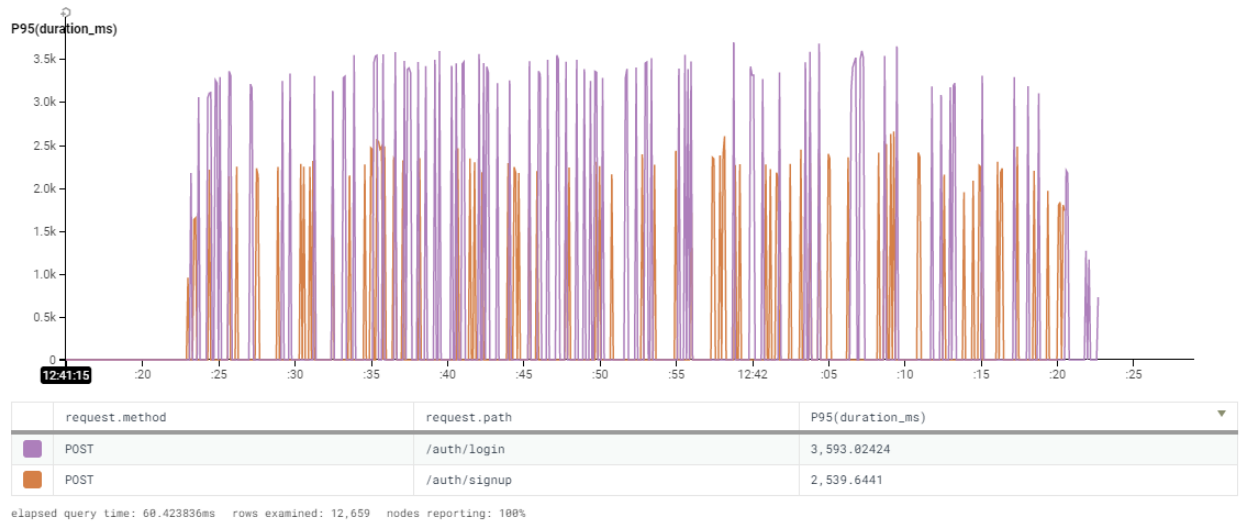


Figure 5.2.5: Post-optimization maximum QPS before degradation graph

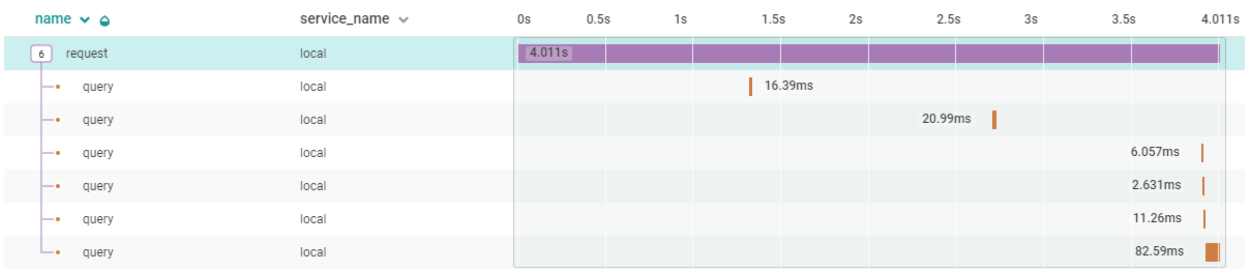
From Figure 5.2.5, we see that the maximum QPS before degradation has doubled from 10 QPS before to 20 QPS. This high of a performance increase was surprising because of how simple the change was, but it confirmed our initial hypothesis of the performance increases of switching to a C++ based library and adding proper indexes in large database tables.



	request.method	request.path	P95(duration_ms)
🟪	POST	/auth/login	3,593.02424
🟧	POST	/auth/signup	2,539.6441

```
elapsed query time: 60.423836ms   rows examined: 12,659   nodes reporting: 100%
```

Figure 5.2.6: Post-optimization maximum QPS before degradation graph



name	service_name	
6 request	local	4.011s
query	local	16.39ms
query	local	20.99ms
query	local	6.057ms
query	local	2.631ms
query	local	11.26ms
query	local	82.59ms

Figure 5.2.7: Post-optimization maximum QPS before degradation graph

From Figure 5.2.6, we see that there is a 76% and 73% reduction in P95 latency for the login endpoint and sign up endpoint, respectively. From the trace in Figure 5.2.7, we see that the SQL query to search for an existing account in the User table has gone down from around 177ms to just about 16 ms, a 94% decrease in time. This is consistent with our understanding of SQL queries; we have reduced the search time from a linear search to a B-tree search. Although the optimization produced great results, we did notice that the latency of a computationally expensive hashing algorithm, indicated by the time gap, is ultimately unavoidable for security purposes.

5.3 Redis Caching of Sessions

Baseline Performance

After optimizing for login and signup, we were ready to switch up the load testing script to test the scalability of linking an external bank account to our service and transferring funds from an external account to an XCurrency internal account. The reason why we selected this particular user scenario was because it is a procedure that every new XCurrency user must perform to begin exchanging currency with others. Specifically, the load script will link a set of Canadian external checking and savings bank accounts from Chase by parsing a hardcoded JSON response normally returned when the user authenticates to their bank accounts through the Plaid API. The account creation process will identify these external accounts as holding Canadian dollars and will also create a XCurrency CAD internal account. Each test user will then transfer $5000 from their external checking account into their internal CAD XCurrency account to simulate the steps required to begin exchanging currency with others through the CAD internal account.

The script makes a total of 10 HTTP requests: accessing the landing page, sign up page, login page, profile page, transfer page, and hitting the signup, login, account creation, transfer balance, and logout endpoints. To determine a new baseline performance, we followed the same process of increasing K6 load intensity but used this new script instead. This produced the following figure.

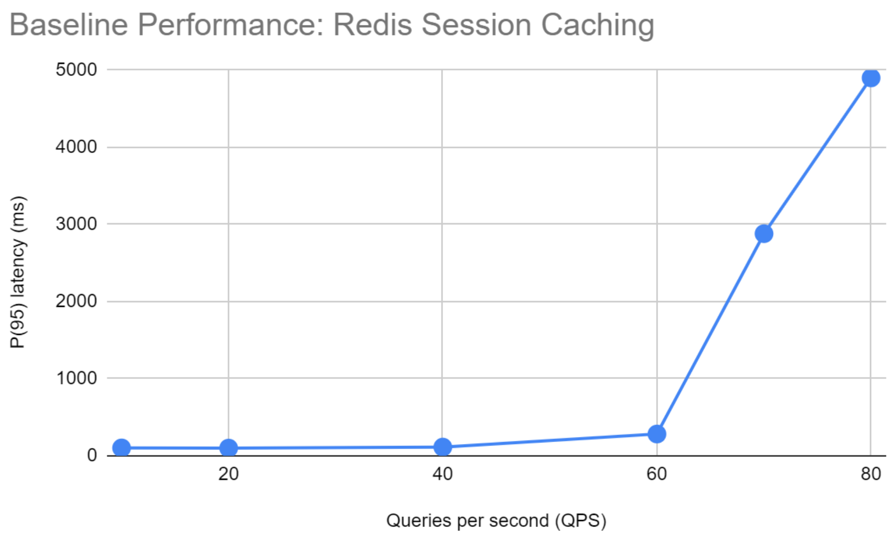
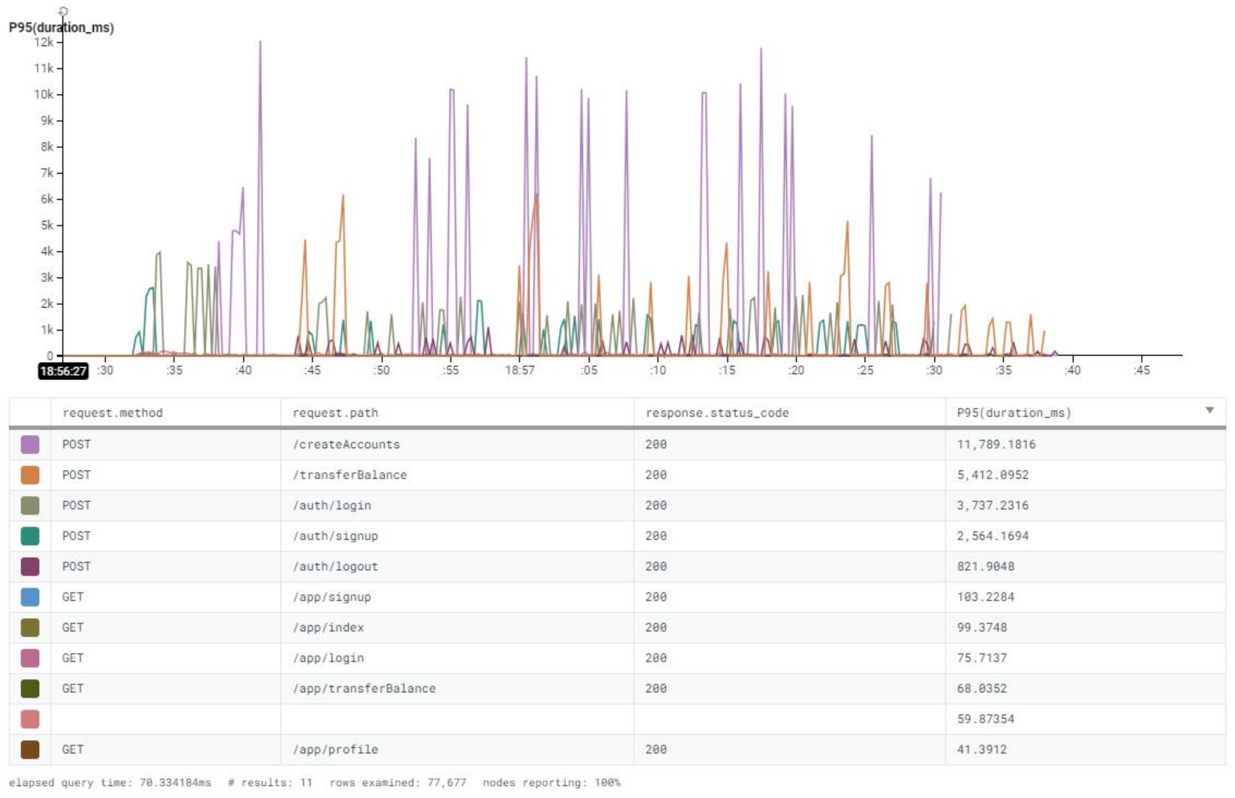


Figure 5.3.1: Pre-optimization QPS before degradation graph for transfer balance load testing

From Figure 5.3.1, we can roughly estimate the QPS to be around 60 before the performance of the server degrades, a threshold defined by a sharp increase in P95 latency. Next, to diagnose possible optimizations, we ran the ramping executor load test type for a maximum of 200 concurrent users and obtained the following Honeycomb results.



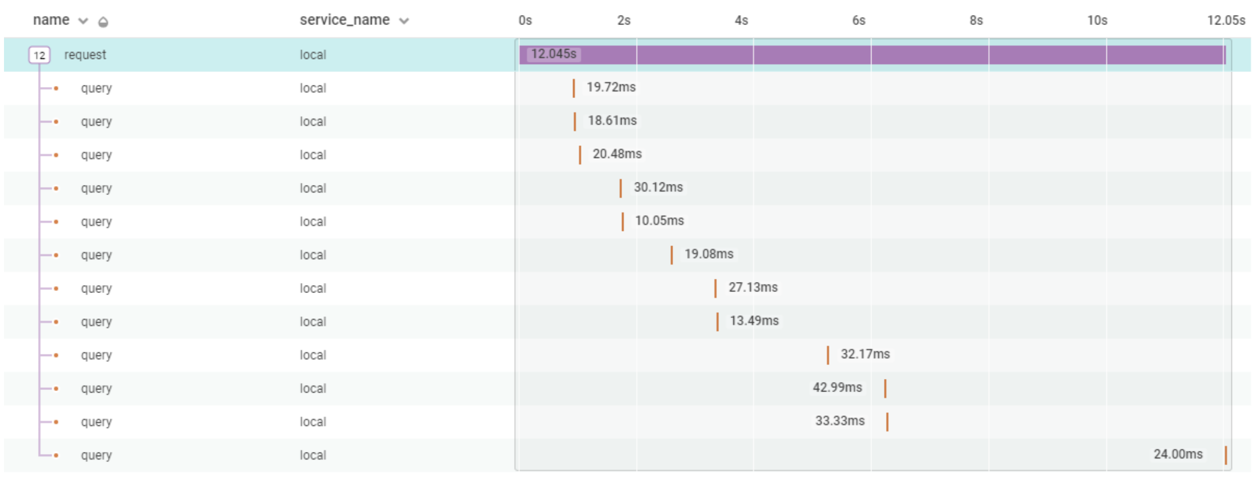
	request.method	request.path	response.status_code	P95(duration_ms) ▼
🟪	POST	/createAccounts	200	11,789.1816
🟧	POST	/transferBalance	200	5,412.0952
🟩	POST	/auth/login	200	3,737.2316
🟩	POST	/auth/signup	200	2,564.1694
🟪	POST	/auth/logout	200	821.9048
🟦	GET	/app/signup	200	103.2284
🟫	GET	/app/index	200	99.3748
🟪	GET	/app/login	200	75.7137
🟩	GET	/app/transferBalance	200	68.0352
🟥				59.87354
🟫	GET	/app/profile	200	41.3912

elapsed query time: 70.334184ms # results: 11 rows examined: 77,677 nodes reporting: 100%

Figure 5.3.2: Pre-optimization Honeycomb P95 latency results

From Figure 5.3.2, we made the following observations about the existing performance of the newly introduced account creation and transfer balance script.

- Individual page fetches, indicated by GET calls to /app/*, were all well under a second and do not appear to require optimization for a target load of 200 concurrent users. This is likely due to the Apollo caching of fetch user calls and the simplistic design of our user interface which primarily contains static elements.
- However, the POST request to add internal and external bank accounts to the database, /createAccounts, was very obviously a bottleneck with 5% of all users experiencing a massive 11.8 second wait time under a load of 200 concurrent users. The P50 value for /createAccounts was also incredibly high -- 9.8 seconds, which indicated that there was a clear underlying issue with the code that affected a majority of the users rather than a special case where a smaller subset of users were affected under load.
- The POST request to /transferBalance also has a high P95 latency, requiring over 5 seconds for 5% of our users. There is a lot of room for optimization here.



name	service_name	duration
12 request	local	12.045s
query	local	19.72ms
query	local	18.61ms
query	local	20.48ms
query	local	30.12ms
query	local	10.05ms
query	local	19.08ms
query	local	27.13ms
query	local	13.49ms
query	local	32.17ms
query	local	42.99ms
query	local	33.33ms
query	local	24.00ms

<u>Figure 5.3.3</u>: Trace of the worst performing case for /createAccounts

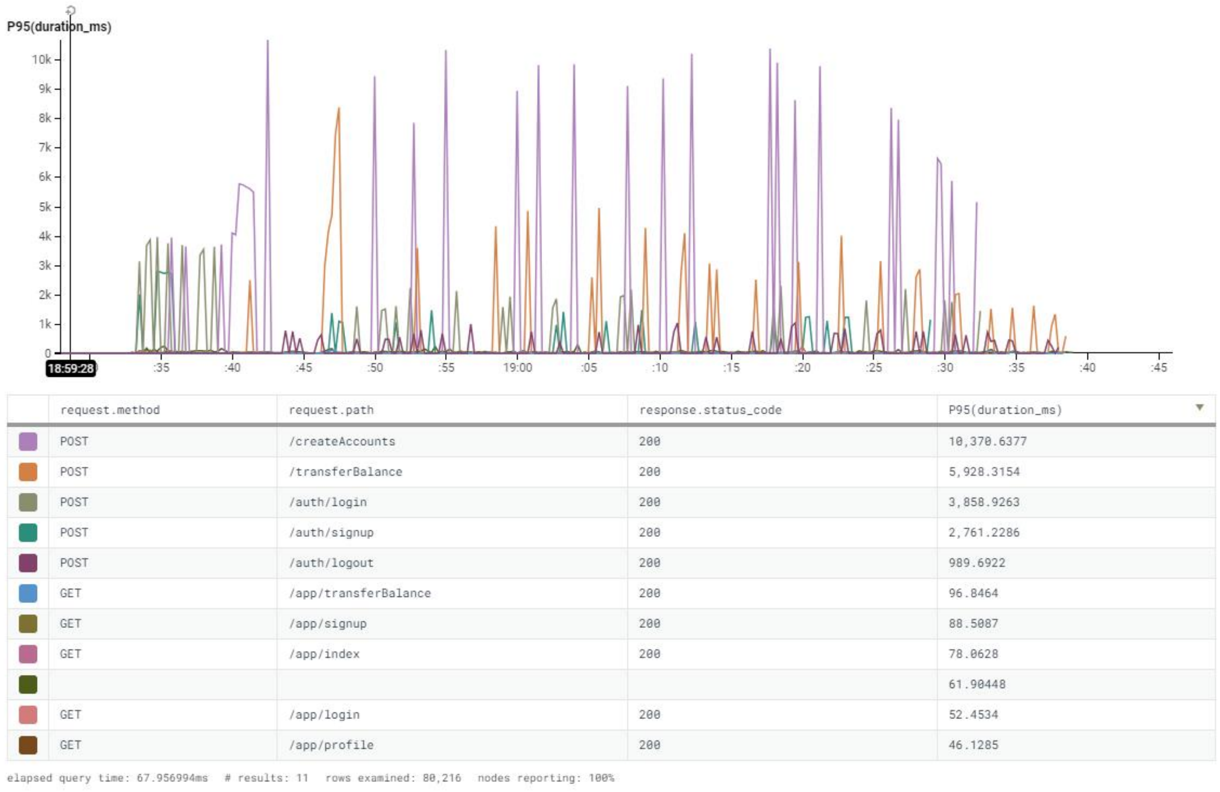
From Figure 5.3.3, we can see that there were a total of 12 database calls performed when calling the /createAccounts endpoint, though the request duration for each seems optimized. The time gap between the last two queries is not so easily explained by any line in the code, which makes this investigation of what to optimize next difficult.

<u>Hypothesis</u>

We observed that there were rather unnecessary calls to retrieve the User object from the Session table that have the potential to be easily cached because the database entries remained static. This seemed like a simple starting point that would prevent 2 calls to the database per endpoint request.

Our proposed solution is to deserialize the user object and cache its value by associating it with its authentication token using Redis. This would be done upon logging in and would be cleared from the Redis cache upon logging out to make room for new cached users.

<u>Implementation of Proposed Solution</u>

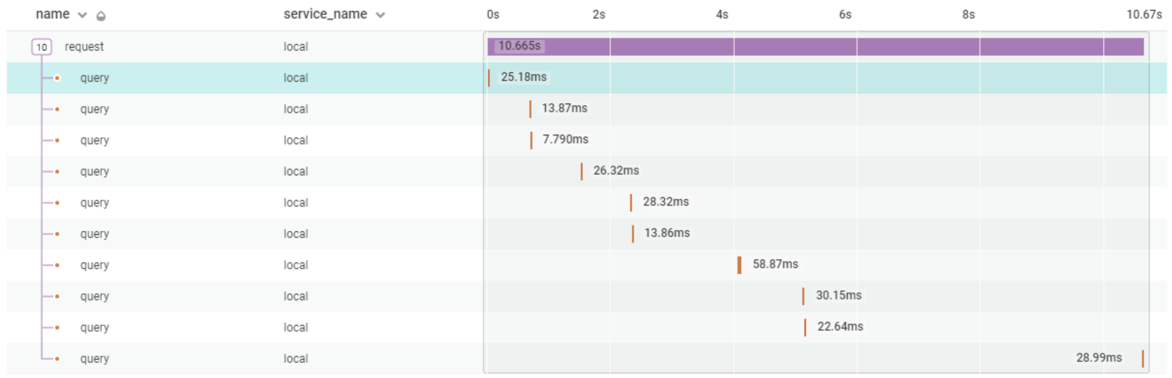


	request.method	request.path	response.status_code	P95(duration_ms) ▼
🟪	POST	/createAccounts	200	10,370.6377
🟧	POST	/transferBalance	200	5,928.3154
🟩	POST	/auth/login	200	3,858.9263
🟩	POST	/auth/signup	200	2,761.2286
🟪	POST	/auth/logout	200	989.6922
🟦	GET	/app/transferBalance	200	96.8464
🟫	GET	/app/signup	200	88.5087
🟪	GET	/app/index	200	78.0628
🟩				61.90448
🟥	GET	/app/login	200	52.4534
🟫	GET	/app/profile	200	46.1285

elapsed query time: 67.956994ms # results: 11 rows examined: 80,216 nodes reporting: 100%

Figure 5.3.4: Post-optimization Honeycomb P95 latency results

From Figure 5.3.4, the P95 latency of the /createAccounts endpoint has decreased by about 12% from 11.7 seconds to 10.3 seconds due to the caching of the two database queries. Other endpoints do not seem to be significantly impacted by the change.



name	service_name	duration
10 request	local	10.665s
query	local	25.18ms
query	local	13.87ms
query	local	7.790ms
query	local	26.32ms
query	local	28.32ms
query	local	13.86ms
query	local	58.87ms
query	local	30.15ms
query	local	22.64ms
query	local	28.99ms

Figure 5.3.5: Post-optimization trace of /createAccounts

Figure 5.3.5 confirms that the Redis caching is implemented correctly; there are now only 10 database operations being performed compared to the 12 database operations previously, and the two Session queries in particular are no longer present.

It appears the overhead of adding Redis caching, which includes increased memory usage, the overhead of maintaining the cache through the removal and insertion of new entries, and the serialization and deserialization of the User object, are less impactful than the performance benefits of avoiding two database lookups in this testing scenario. There does not seem to be a noticeable increase in server time initialization either when establishing a connection to the local Redis server, so the optimization appears to have a net benefit on our system in both runtime performance and developer convenience.

5.4 Performing Parallel SQL Queries

Baseline Performance

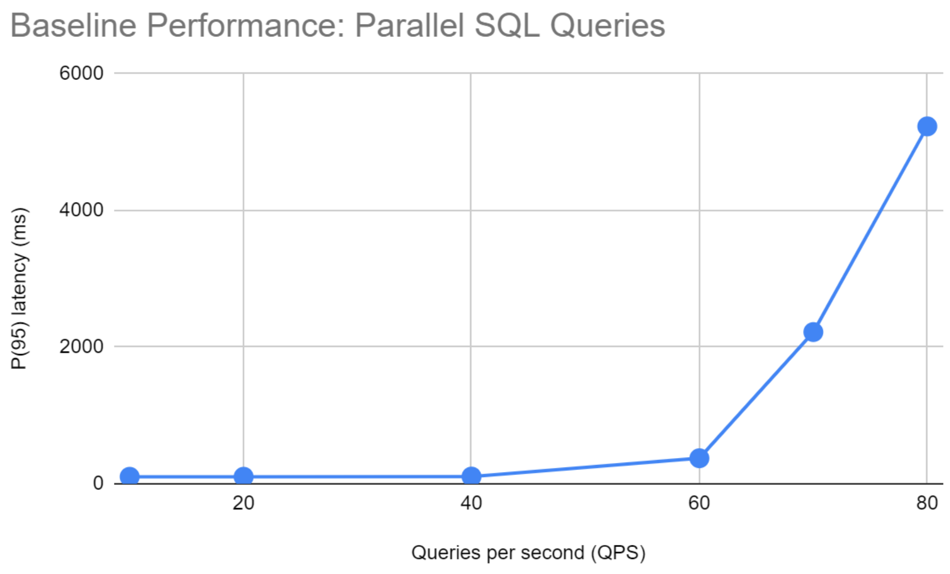
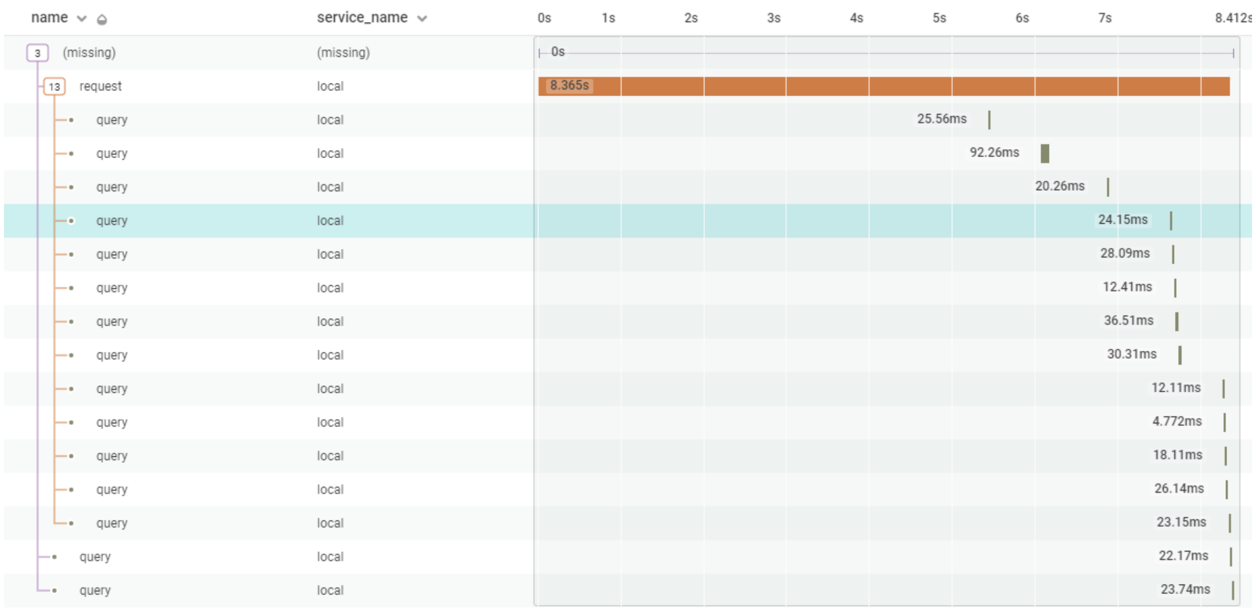


Figure 5.4.1: Baseline performance after adding Redis caching

After adding the Redis caching of user sessions to our baseline performance, we unfortunately did not see a noticeable increase in the maximum QPS before service degradation, as shown in the plotted P95 vs. QPS load test results above in Figure 5.4.1. This may be due to the account creation and balance transfer testing script only having a single endpoint that utilizes the cache effectively, the /createAccounts endpoint. However, our core algorithm, which will be closely examined in Section 5.7, will have more endpoints that use caching, so we predict the benefits will be more valuable for the algorithm.



name	service_name	duration
3 (missing)	(missing)	0s
13 request	local	8.365s
query	local	25.56ms
query	local	92.26ms
query	local	20.26ms
query	local	24.15ms
query	local	28.09ms
query	local	12.41ms
query	local	36.51ms
query	local	30.31ms
query	local	12.11ms
query	local	4.772ms
query	local	18.11ms
query	local	26.14ms
query	local	23.15ms
query	local	22.17ms
query	local	23.74ms

Figure 5.4.2: Trace of one of the worser performing /transferBalance runtimes

The performance of the overall testing script is dominated by the two high latency endpoints, /createAccounts and /transferBalance, so to be more efficient, we prioritized optimizations that would target the two endpoints. From the trace of the worst case runtimes of /createAccounts and /transferBalance shown in Figure 5.3.5 and Figure 5.4.2, we observed that the two endpoints are primarily database heavy, and the cause for high latency seems to be from the many small time gaps between queries rather than the speed of the individual queries themselves.

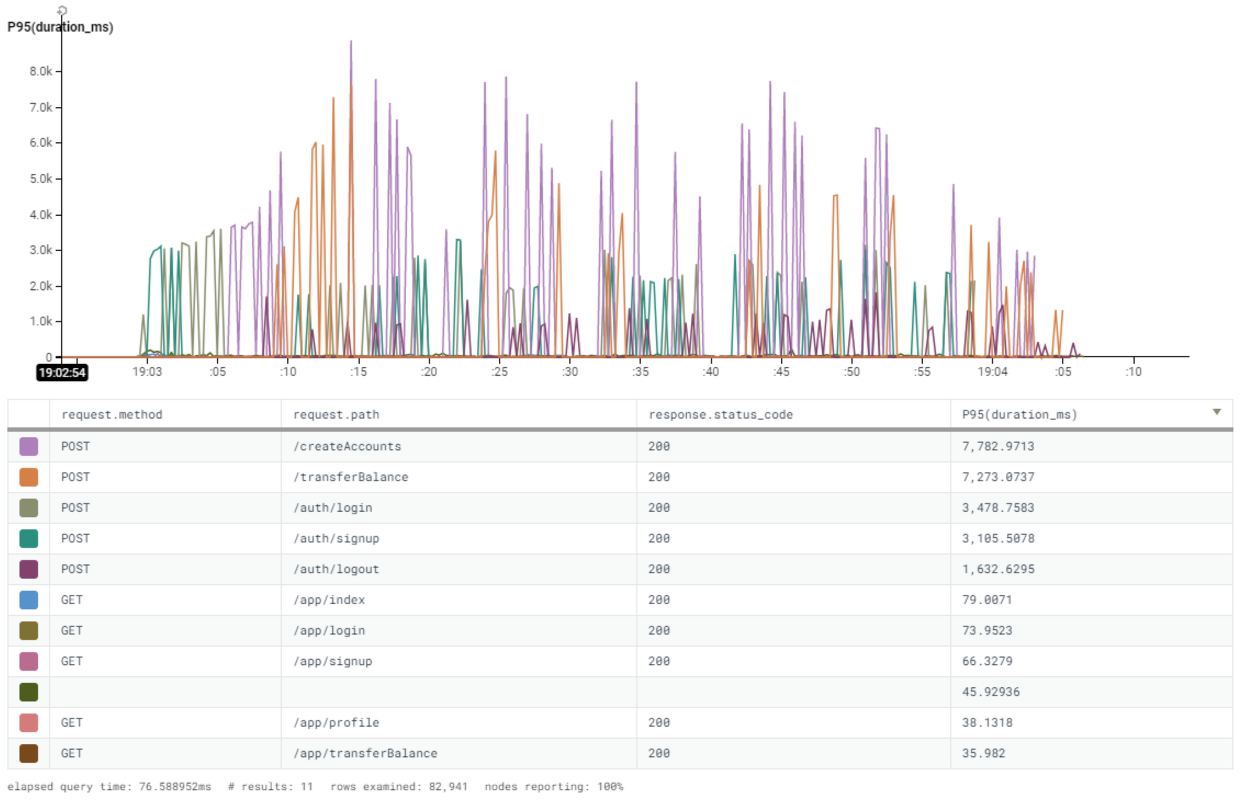
Hypothesis

The first hypothesis for the reason behind the time gaps was identifying that we had areas of code where we were inefficiently using ‘await’ preemptively on database operations when we could have been using Promise.all(). By executing multiple queries at once and delaying the resolution of the response until they were absolutely needed, we could better utilize the parallelism of our MySQL database and reduce the overall latency of each request.

Our second hypothesis for the “invisible” operations that were occurring between each database query was overhead introduced by TypeORM. The login, signup, account creation, and balance transfer endpoints were all using the abstractions provided by TypeORM table classes to retrieve, insert, and update database entries. The overhead of mapping MySQL query results into these TypeORM classes may be contributing to the increased latency of each query.

We wanted to measure the effects of each change separately, so our proposed solution to test the first hypothesis is to refactor both endpoints to use Promise.all() whenever possible on a series of SQL operations. An example of this in /createAccount was to change the logic of creating external and internal accounts separately using await into a Promise.all that created both internal and external accounts in parallel.

<u>Implementation of Proposed Solution</u>

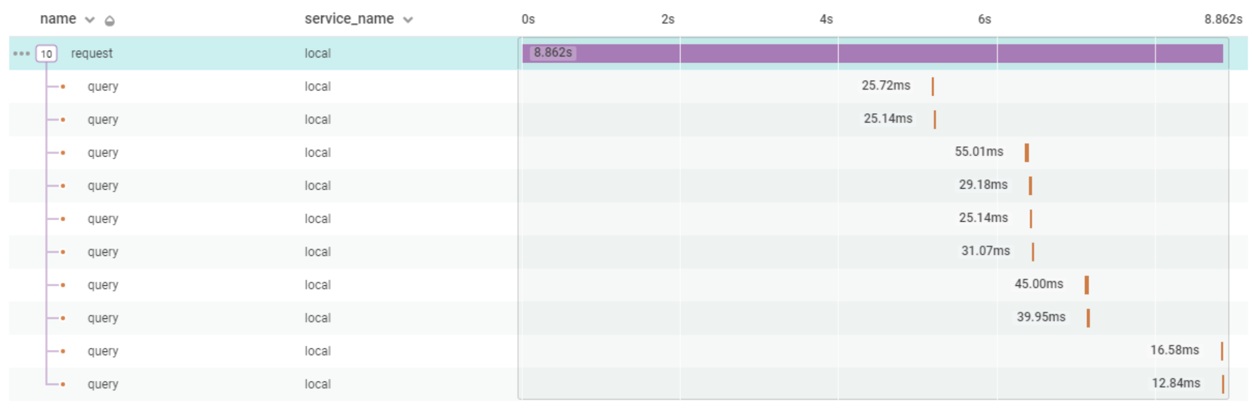


	request.method	request.path	response.status_code	P95(duration_ms)
🟪	POST	/createAccounts	200	7,782.9713
🟧	POST	/transferBalance	200	7,273.0737
🟫	POST	/auth/login	200	3,478.7583
🟩	POST	/auth/signup	200	3,105.5078
🟪	POST	/auth/logout	200	1,632.6295
🟦	GET	/app/index	200	79.0071
🟫	GET	/app/login	200	73.9523
🟪	GET	/app/signup	200	66.3279
🟩				45.92936
🟥	GET	/app/profile	200	38.1318
🟫	GET	/app/transferBalance	200	35.982

```
elapsed query time: 76.588952ms   # results: 11   rows examined: 82,941   nodes reporting: 100%
```

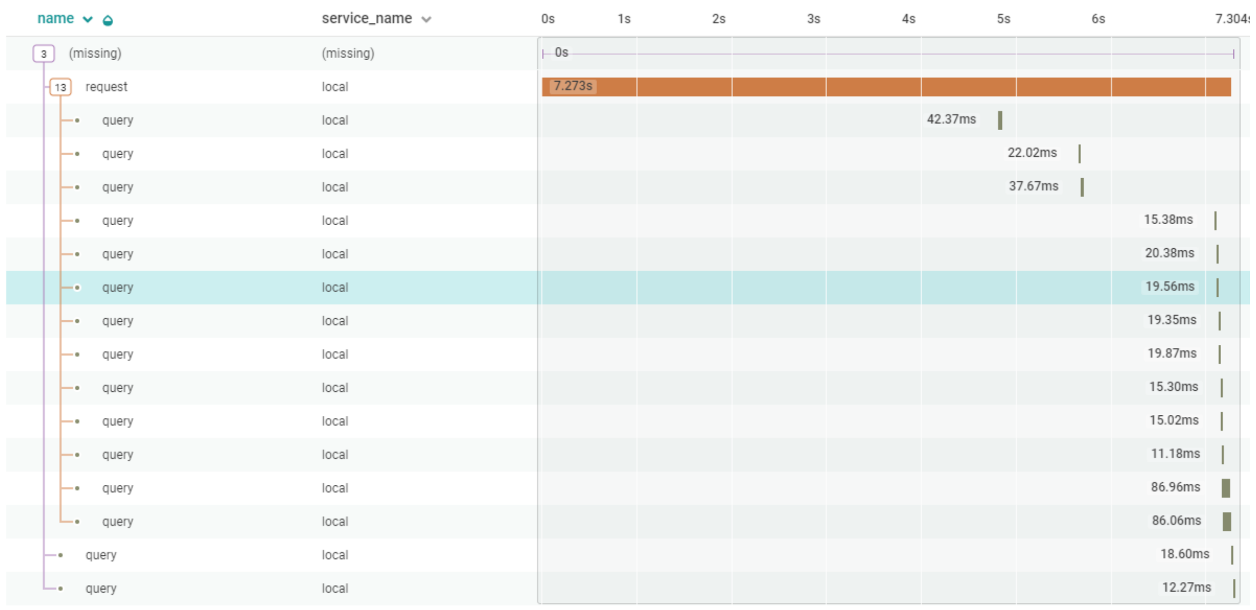
<u>Figure 5.4.3</u>: Post-optimization Honeycomb P95 latency results

From Figure 5.4.3, we see that there is a 25% decrease in P95 latency for the /createAccounts endpoint, from 10.3 seconds to 7.7 seconds. This result was expected because we were able to have two sets of two queries run in parallel in this endpoint by delaying the resolution of the queries. However, we cannot quite explain why the P95 latency for the /transferBalance endpoint went up by 22%, from 5.9 seconds to 7.2 seconds. We suspect that it may be due to the SQL connection limit being set too low, which hindered the performance of the endpoint even as its parallelism increased due to the cap of 5 TypeORM connections.



name	service_name	0s – 8.862s
10 request	local	8.862s
query	local	25.72ms
query	local	25.14ms
query	local	55.01ms
query	local	29.18ms
query	local	25.14ms
query	local	31.07ms
query	local	45.00ms
query	local	39.95ms
query	local	16.58ms
query	local	12.84ms

Figure 5.4.4: Post-optimization one of the worser trace of /createAccounts



name	service_name	0s – 7.304s
3 (missing)	(missing)	0s
13 request	local	7.273s
query	local	42.37ms
query	local	22.02ms
query	local	37.67ms
query	local	15.38ms
query	local	20.38ms
query	local	19.56ms
query	local	19.35ms
query	local	19.87ms
query	local	15.30ms
query	local	15.02ms
query	local	11.18ms
query	local	86.96ms
query	local	86.06ms
query	local	18.60ms
query	local	12.27ms

Figure 5.4.5: Post-optimization one of the worser traces of /transferBalance

From the two traces shown in Figure 5.4.4 and Figure 5.4.5 and by comparing against the pre-optimization traces in Figure 5.3.3 and Figure 5.4.2, it is clear that by parallelizing multiple SQL queries, we were able to speed up each set of SQL operations because of their now overlapping operation intervals. However, the consequence of such parallel queries may have added pressure onto the MySQL connection pool, which we kept constant throughout at its default values of 5 for TypeORM and 15 total.

5.5 Optimize SQL calls with Raw SQL instead of TypeORM

Baseline Performance

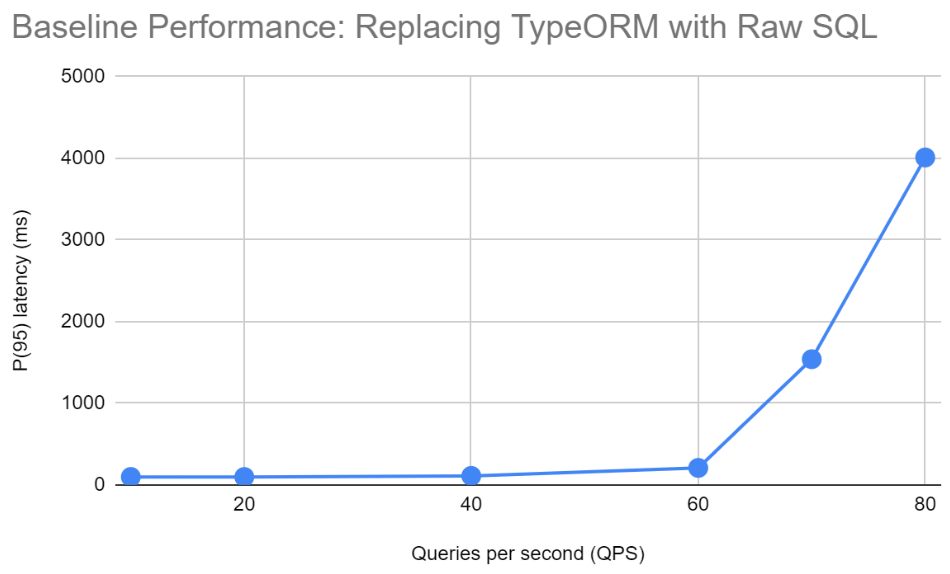


Figure 5.5.1: Baseline performance after parallelizing SQL operations

After adding the changes to parallelize SQL operations, there appears to be a tiny drop in P95 latency across all levels of QPS, particularly at 80 QPS, but the maximum QPS is still 60. This does make sense considering the mixed performance results in P95 latency as explained in Section 5.4 due to potentially too low of a connection limit for the parallelized TypeORM database operations.

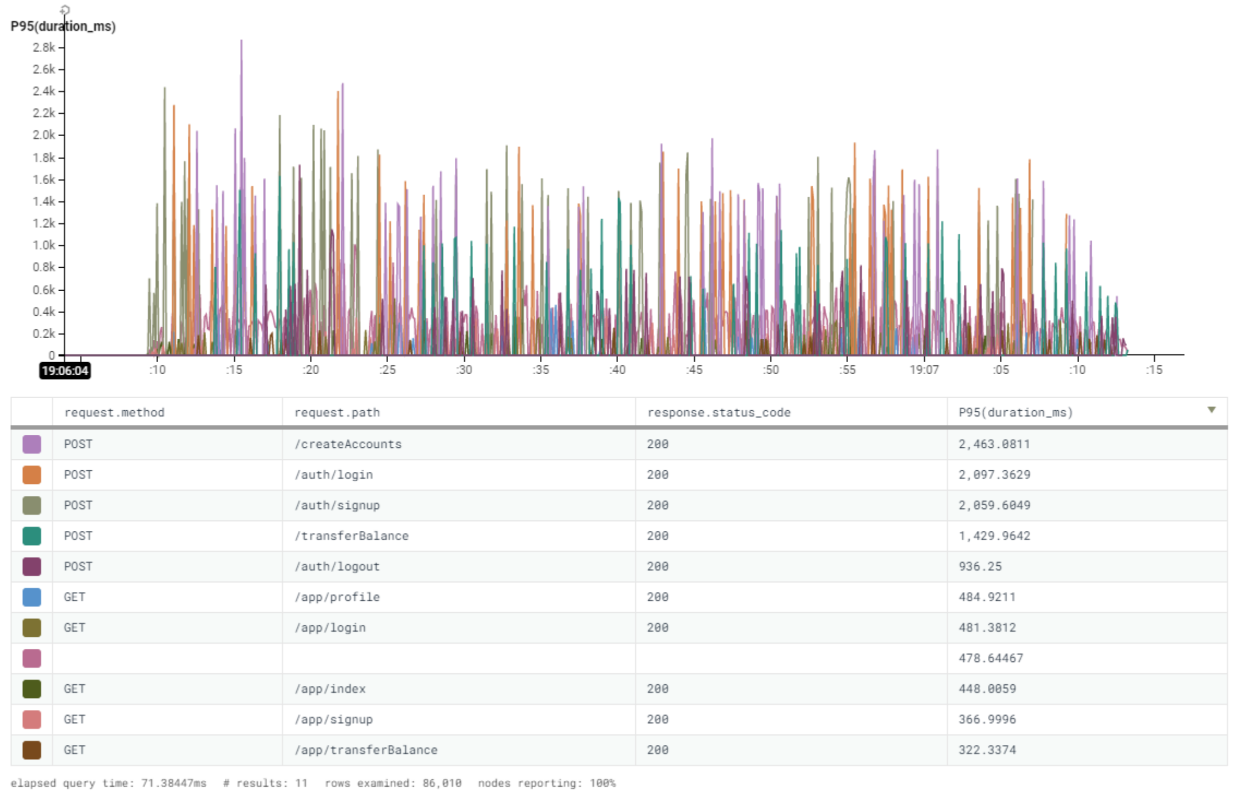
Recall our second hypothesis for the "invisible" operations that were occurring between each database query in /createAccounts and /transferBalance may be attributed to the TypeORM overhead. We will now proceed to test this hypothesis.

Hypothesis

______Our proposed solution to test our hypothesis is to replace TypeORM queries with raw SQL queries, thereby reducing the potential of TypeORM overhead. Since raw SQL provides low-level database interactions and TypeORM creates an object for each table in the database, we predicted that converting most of our queries could promote efficiency and remove the extra level of indirection created with TypeORM. Also, due to the developer-friendly nature of TypeORM, the representation of higher-level operations using the limited set of SQL primitives may be inefficient, as may be the case for updating objects that have been modified using the save() function, for instance.

Implementation of Proposed Solution

One issue we ran into while load testing the optimization from TypeORM to raw SQL operations was that the server seemingly kept freezing without any error messages appearing in the console. It turned out that we were finally maxing out the 15 maximum connection limit allocated for the SQL connection pool. To test the changes, we arbitrarily chose 130 total SQL connections and 20 TypeORM connections but planned to optimize the connection limit more thoroughly next.



	request.method	request.path	response.status_code	P95(duration_ms)
🟪	POST	/createAccounts	200	2,463.0811
🟧	POST	/auth/login	200	2,097.3629
🟩	POST	/auth/signup	200	2,059.6049
🟩	POST	/transferBalance	200	1,429.9642
🟪	POST	/auth/logout	200	936.25
🟦	GET	/app/profile	200	484.9211
🟫	GET	/app/login	200	481.3812
🟪				478.64467
🟩	GET	/app/index	200	448.0059
🟥	GET	/app/signup	200	366.9996
🟫	GET	/app/transferBalance	200	322.3374

```
elapsed query time: 71.38447ms   # results: 11   rows examined: 86,010   nodes reporting: 100%
```

Figure 5.5.2: Post-optimization Honeycomb P95 latency results

From Figure 5.5.2, we see a 68% decrease in P95 latency in the /createAccounts endpoint, from 7.7 seconds to 2.4 seconds. We also see an 80% decrease in P95 latency in the /transferBalance endpoint, from 7.2 seconds to 1.4 seconds. The large drop in latency for the transfer balances endpoint result is most likely partially due to the increase in the SQL connection limit, which we hypothesized would be throttled due to increased parallelism.

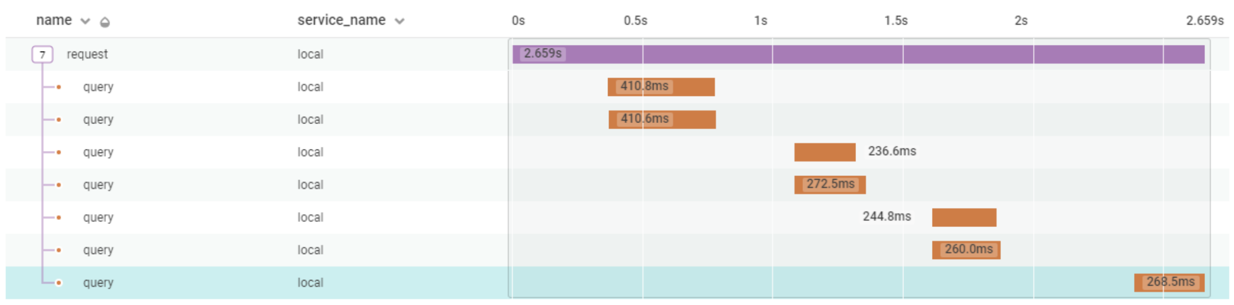


Figure 5.5.3: Post-optimization Honeycomb trace of /createAccounts

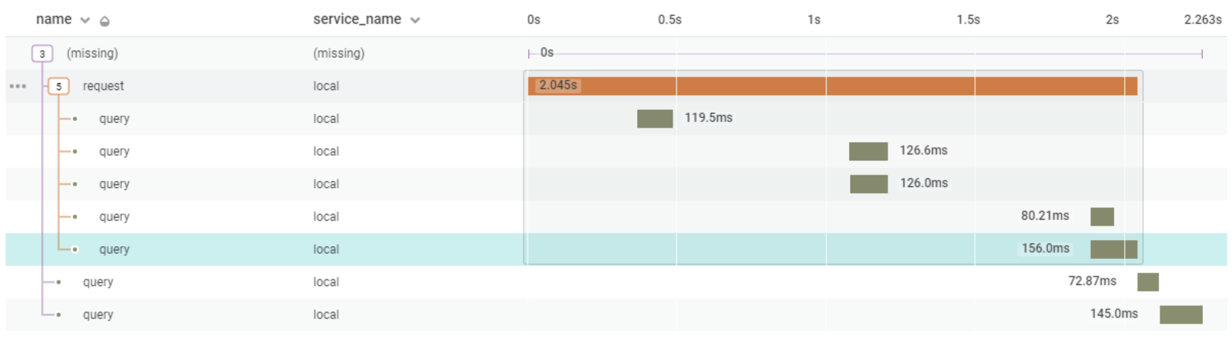


Figure 5.5.4: Post-optimization Honeycomb trace of /transferBalance

One aspect of these two traces that stands out in particular is that in Figure 5.5.4, the final two database operations of COMMIT and SET AUTOCOMMIT = 1 are no longer counted towards the latency of the endpoint because they are performed by SQL after the endpoint response has been sent to the user. Another important aspect to point out is the increase in SQL operation times in Figure 5.5.3, which we suspect may be due to inaccurate reporting from before due to TypeORM abstracting the process.

5.6 Connection Pool Optimizations

Baseline Performance

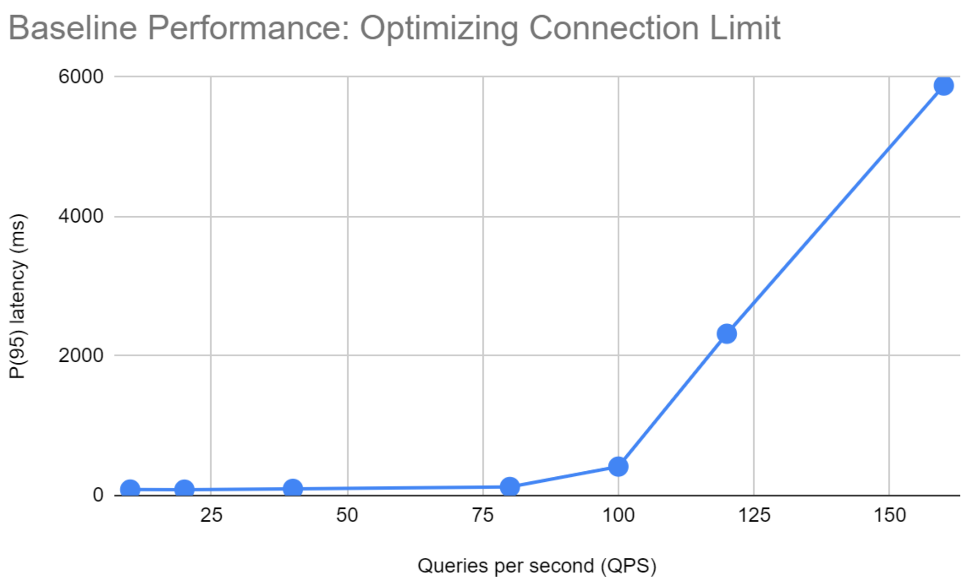


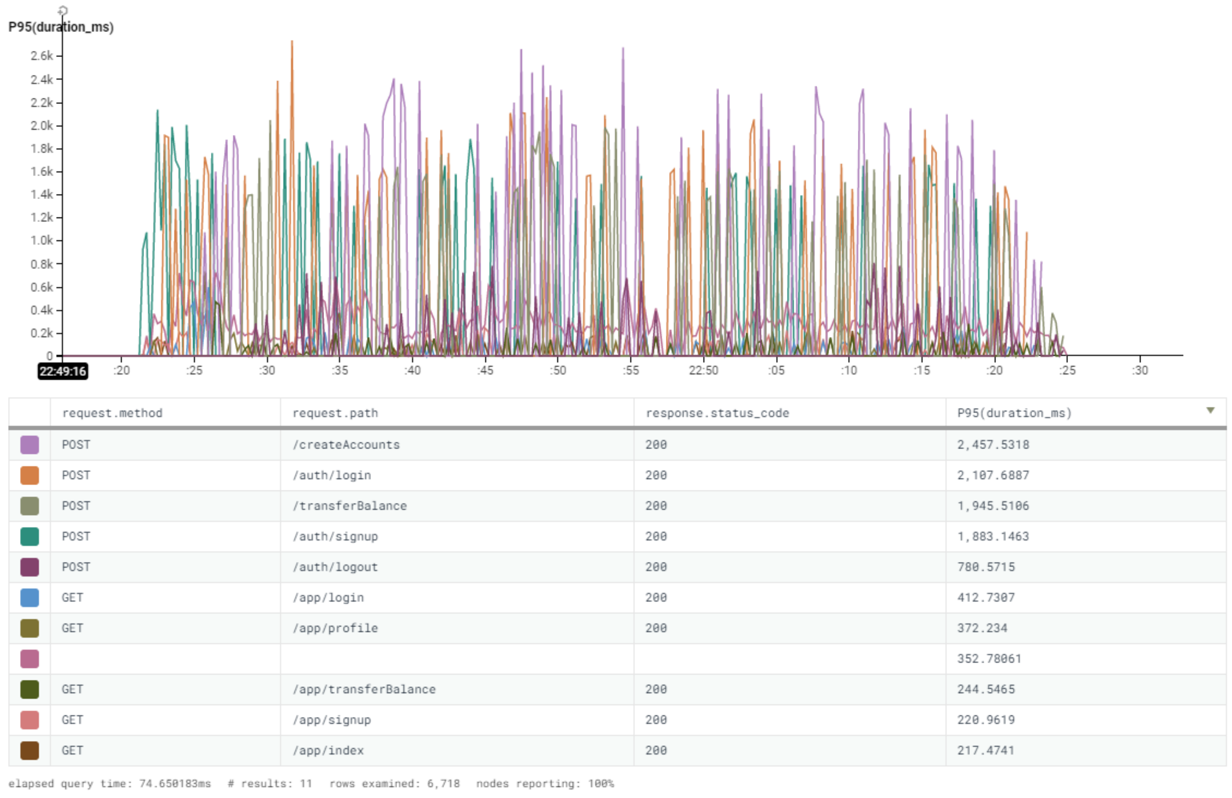
Figure 5.6.1: Baseline performance of our application prior to implementing connection pool and connection limit related optimizations.

As a result of the TypeORM to raw SQL optimization, Figure 5.6.1 shows that there is a 66.7% increase in the maximum number of QPS from 60 to 100 QPS before degraded performance. Some of the routes with the worst latencies include /createAccounts, /auth/login, /auth/signup, and /transferBalance. We predict that this is due to these routes using more database transactions compared to the other routes of our application since these endpoints are searching and inserting in the database frequently.

From performance traces, which show at what point timeouts occur due to a MySQL database connection not being available, we can see that the latency is coming from a database transaction waiting, for a rather lengthy time, for a connection to be available to perform a database operation.

Hypothesis

Since we have now changed from type ORM to raw SQL queries, added parallelized queries, implemented caching, added Redis, changed to better libraries, and fixed the ApolloClient's server-side rendering configuration, we came to the understanding that the connection pool for the database can affect our app's performance and latency. We think some of the queries we are trying to perform are waiting for other connections to free up due to the very limited connection limit currently implemented in the app. The supporting evidence for this lies in the fact that with a low connection limit, we are finding that the prior optimizations of parallel queries and converting several type ORM queries to raw SQL leads to an issue when doing heavy load testing in that a lot of the database transactions timeout without performing their task. This indicates that the connection pool and its limits are simply not large enough to properly handle this scale. Prior to this proper optimization, we arbitrarily chose new connection pool limits for the database and type ORM as mentioned above, but this optimization focuses on finding the optimal connection limit for our intended use case figured with having a connection pool so big that most of the connections are idle and adding overhead. Considering that we want to handle 200 users, meaning we want to accomplish 200 queries per second without degraded performance, and each query is in the database for about 200-250 milliseconds based on the above analyses, this means the database connection pool should support a connection limit of 50 - 60 connections.

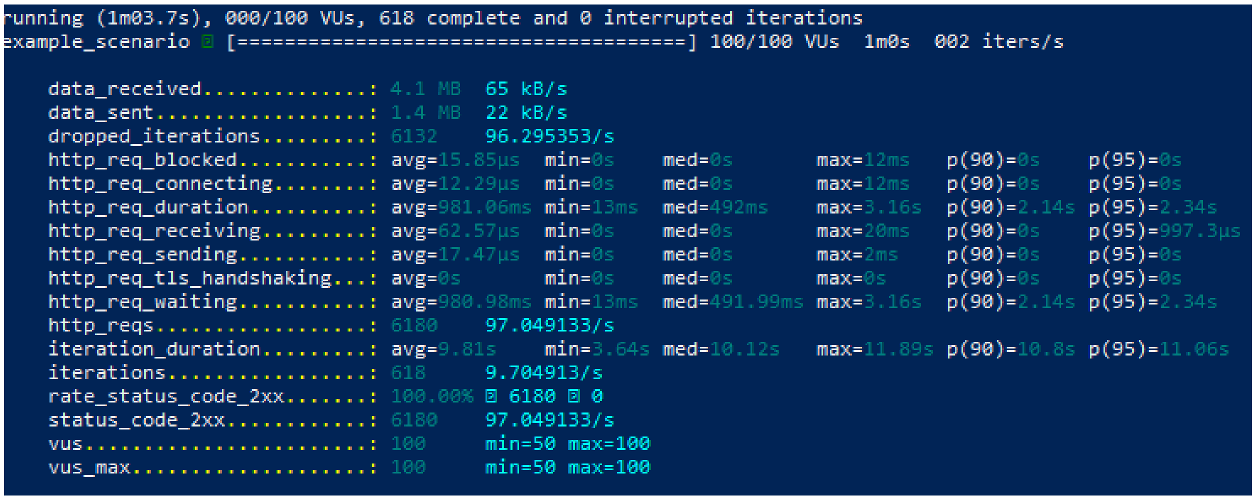


	request.method	request.path	response.status_code	P95(duration_ms)
🟪	POST	/createAccounts	200	2,457.5318
🟧	POST	/auth/login	200	2,107.6887
🟩	POST	/transferBalance	200	1,945.5106
🟩	POST	/auth/signup	200	1,883.1463
🟪	POST	/auth/logout	200	780.5715
🟦	GET	/app/login	200	412.7307
🟫	GET	/app/profile	200	372.234
🟪				352.78061
🟩	GET	/app/transferBalance	200	244.5465
🟥	GET	/app/signup	200	220.9619
🟫	GET	/app/index	200	217.4741

elapsed query time: 74.650183ms # results: 11 rows examined: 6,718 nodes reporting: 100%

Figure 5.6.2: Honeycomb P95 duration and the color code for each of the routes in our application. https://ui.honeycomb.io/xcurrency/datasets/bespin/result/3KRvFwPUzy2

Implementation of Proposed Solution



```
running (1m03.7s), 000/100 VUs, 618 complete and 0 interrupted iterations
example_scenario ✓ [======================================] 100/100 VUs  1m0s  002 iters/s

     data_received..............: 4.1 MB  65 kB/s
     data_sent..................: 1.4 MB  22 kB/s
     dropped_iterations.........: 6132    96.295353/s
     http_req_blocked...........: avg=15.85µs  min=0s    med=0s        max=12ms   p(90)=0s    p(95)=0s
     http_req_connecting........: avg=12.29µs  min=0s    med=0s        max=12ms   p(90)=0s    p(95)=0s
     http_req_duration..........: avg=981.06ms min=13ms  med=492ms     max=3.16s  p(90)=2.14s p(95)=2.34s
     http_req_receiving.........: avg=62.57µs  min=0s    med=0s        max=20ms   p(90)=0s    p(95)=997.3µs
     http_req_sending...........: avg=17.47µs  min=0s    med=0s        max=2ms    p(90)=0s    p(95)=0s
     http_req_tls_handshaking...: avg=0s       min=0s    med=0s        max=0s     p(90)=0s    p(95)=0s
     http_req_waiting...........: avg=980.98ms min=13ms  med=491.99ms  max=3.16s  p(90)=2.14s p(95)=2.34s
     http_reqs..................: 6180    97.049133/s
     iteration_duration.........: avg=9.81s    min=3.64s med=10.12s    max=11.89s p(90)=10.8s p(95)=11.06s
     iterations.................: 618     9.704913/s
     rate_status_code_2xx.......: 100.00% ✓ 6180 ✗ 0
     status_code_2xx............: 6180    97.049133/s
     vus........................: 100     min=50 max=100
     vus_max....................: 100     min=50 max=100
```

Figure 5.6.3: k6 testing for the connection limit optimizations.

We found that for this scenario, we still need some connections for the type ORM queries since there are still some tasks using it, so we chose to make the database connection pool have a connection limit of 55 with type ORM having a connection limit of 5.

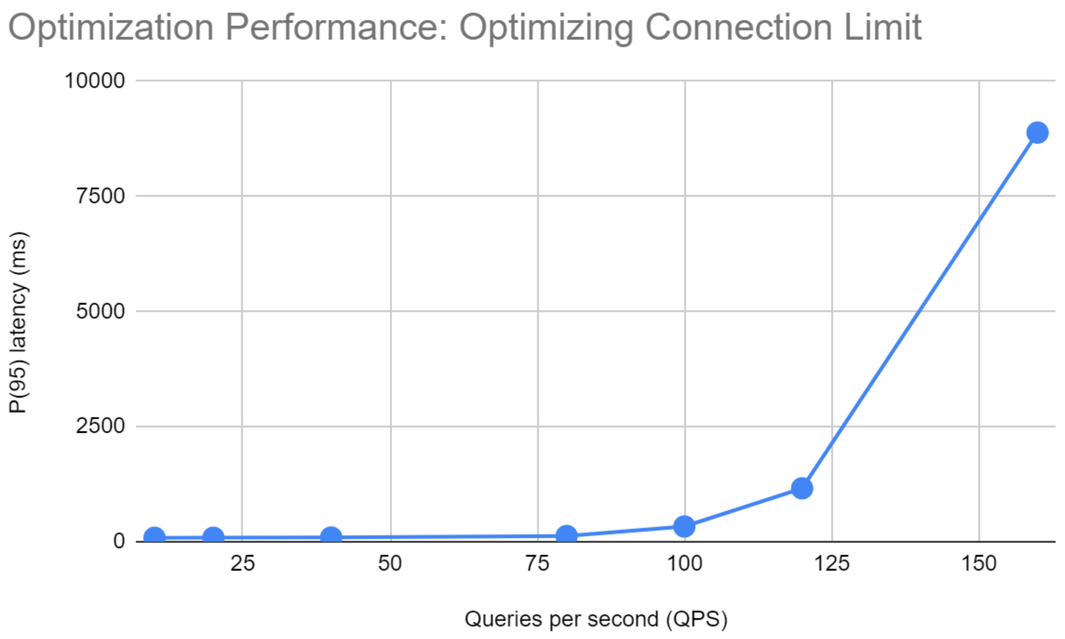


Figure 5.6.4: Performance after completing the connection pool and connection limit optimizations.

As predicted, optimizing the connection pool is leading to better performance in our applications since the load tests no longer crash due to the timeouts mentioned above, and our application can handle a higher number of queries per second with less degraded performance than before. This improvement is due to having more connections in the database connection pool, meaning MySQL transactions are not stuck waiting for a connection to be available to perform the needed database operations. This ensures that the timeouts and resulting crashes no longer occur as they did prior. Thus, this connection pool optimization increases performance, provides better latency, and even increases throughput. However, despite seeing some improvement here, it is not nearly as much as we predicted because as evidenced in the graphs above, there are only subtle performance improvements instead of drastic ones, and the exact percentage of improvement is only 1.8% in terms of the QPS prior to degraded performance. Also, for 100 QPS, we originally had a P95 latency of 415 ms, and after the optimization, 100 QPS has a latency of 331.04 ms, which is a 20% decrease in latency. Regardless, choosing connection pool limits was done more strategically compared to picking an arbitrary number as we did above to prevent the load tests crashing, so this could indicate why the performance results are not as drastic as we hoped.

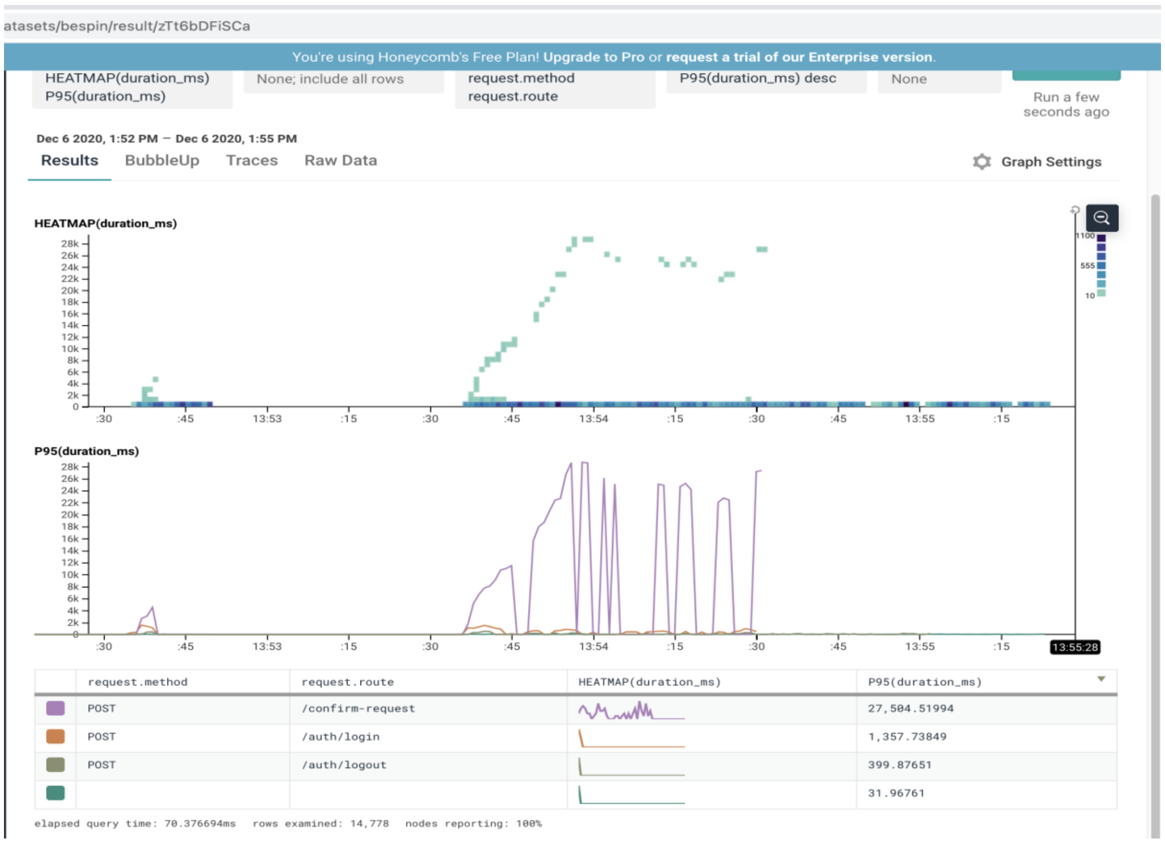
Due to the nature of this change, this fix has not created new problems because issues with raw MySQL or parallelized queries would have shown themselves long before this change considering the previous connection pool size for the database. Regardless, there are no issues with this optimized version either in terms of the other changes made and features developed. However, this does show us that we need to investigate optimizing other parts of our application that aren't necessarily database related.

5.7 Background Process for Algorithm

Baseline Performance

Post ('/confirm-request') endpoint is the most computationally expensive endpoint in the program. The endpoint takes an exchange request from the client and runs an algorithm to find a match in the DB. It has a lot of sql queries wrapped in a transaction to obtain ACID property. One of the strategies to cut down the latency is to run the matching algorithm in a background process instead of having it run based on user request. Furthermore, this will help to ensure that the number of matching algorithms run per time unit will not be proportional to the number of requests created by users.

For our load testing scenarios, we did the following: ramped up execution : {target: 200, duration: 30s}, {target: 0, duration 30s, pre allocated users: 50, maximum users : 200, and as many iterations as possible for each VU.


atasets/bespin/result/zTt6bDFiSCa

You're using Honeycomb's Free Plan! Upgrade to Pro or request a trial of our Enterprise version.

HEATMAP(duration_ms) P95(duration_ms) | None; include all rows | request.method request.route | P95(duration_ms) desc | None

Run a few seconds ago

Dec 6 2020, 1:52 PM – Dec 6 2020, 1:55 PM

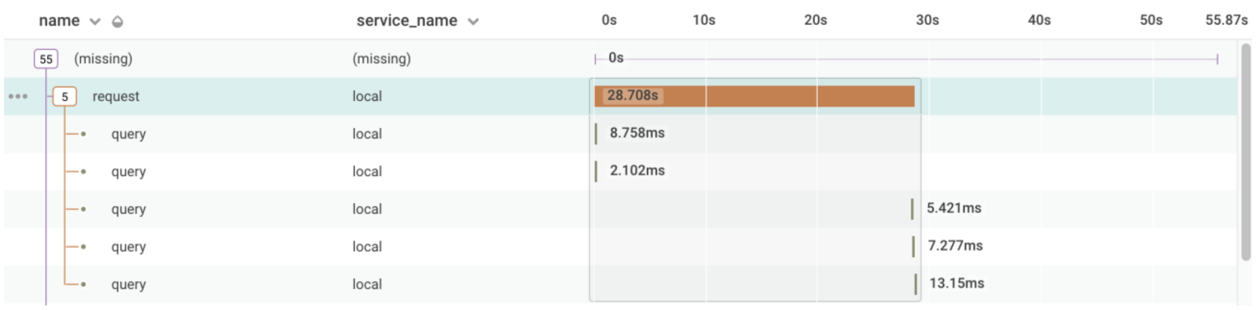
Results BubbleUp Traces Raw Data

Graph Settings

	request.method	request.route	HEATMAP(duration_ms)	P95(duration_ms)
🟪	POST	/confirm-request		27,504.51994
🟧	POST	/auth/login		1,357.73849
🟩	POST	/auth/logout		399.87651
🟩				31.96761

```
elapsed query time: 70.376694ms   rows examined: 14,778   nodes reporting: 100%
```

Figure 5.7.1: Heatmap and Graph (P95) for creating exchange request endpoint.
https://ui.honeycomb.io/xcurrency/datasets/bespin/result/zTt6bDFiSCa



name	service_name	0s 10s 20s 30s 40s 50s 55.87s
55 (missing)	(missing)	0s
5 request	local	28.708s
query	local	8.758ms
query	local	2.102ms
query	local	5.421ms
query	local	7.277ms
query	local	13.15ms

Figure 5.7.2: Performance trace of a 'confirm-request' with the worst latency.

Hypothesis

The main reason that the Post('/confirm-request') P95 duration is so high is the exchange currency algorithm is wrapped in a SQL transaction to ensure all the requests are exchanged with ACID properties (Atomic, Consistency, Isolation, Durable).

As evidenced, in this highest delay of response trace, the queries after the first two queries which are wrapped in a transaction have to wait for at least 28 seconds before they can begin. The 28 seconds delay is also contributed by the node server busy computing previous exchange algorithms and handling other requests. To show how much transaction property has impacted our performance, we plotted the graph below:

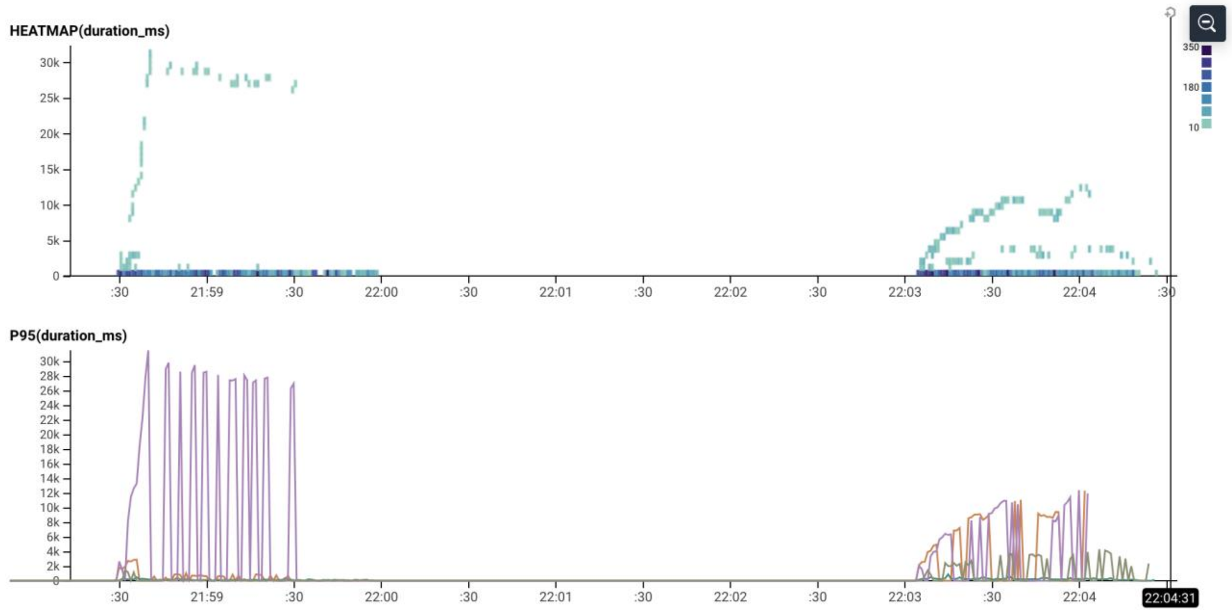


Figure 5.7.6: Visualizing the delay caused by the transaction.
https://ui.honeycomb.io/xcurrency/datasets/bespin/result/egHB6a3iYWh

Since NodeJS is only single threaded, the server is not good at handling heavy computational requests, so we placed the background process on a separate server. The transaction wrapped around the algorithm that is triggered by POST (‘confirm-request’) endpoint is a huge bottleneck that delays the response on average by 20 seconds.

In order to have performance improvements, we implemented the following optimizations: eager response, placing the matching algorithm in the background process, raw SQL statement and parallel queries, and placing the algorithm on a separate server. For the baseline performance, the longest response could have been 55.87 seconds. The reason that it is decreased to 28 seconds is because the server responds immediately to the client without having to wait for the whole algorithm to finish.

However, the interval of the background process can affect the overall performance of the server. The above graph is generated by setting the interval to be 3 seconds. Generally, a shorter time interval leads to longer latency of the overall server. However, a shorter interval also means a match is generated faster for the purpose of user’s convenience.

Implementation of Proposed Solution

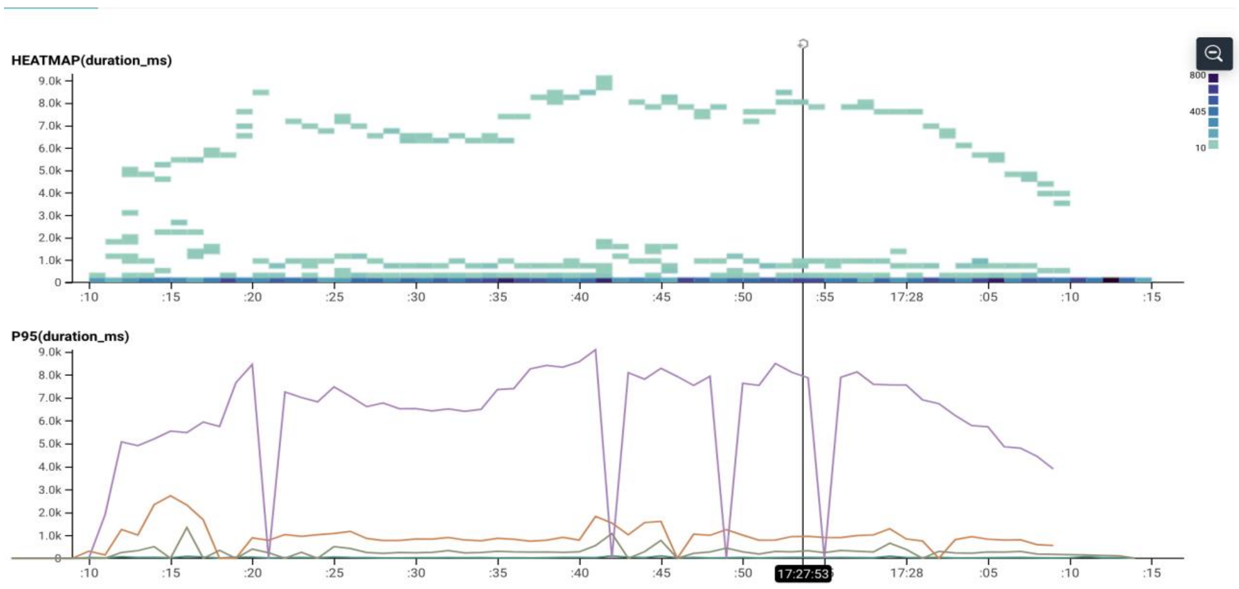
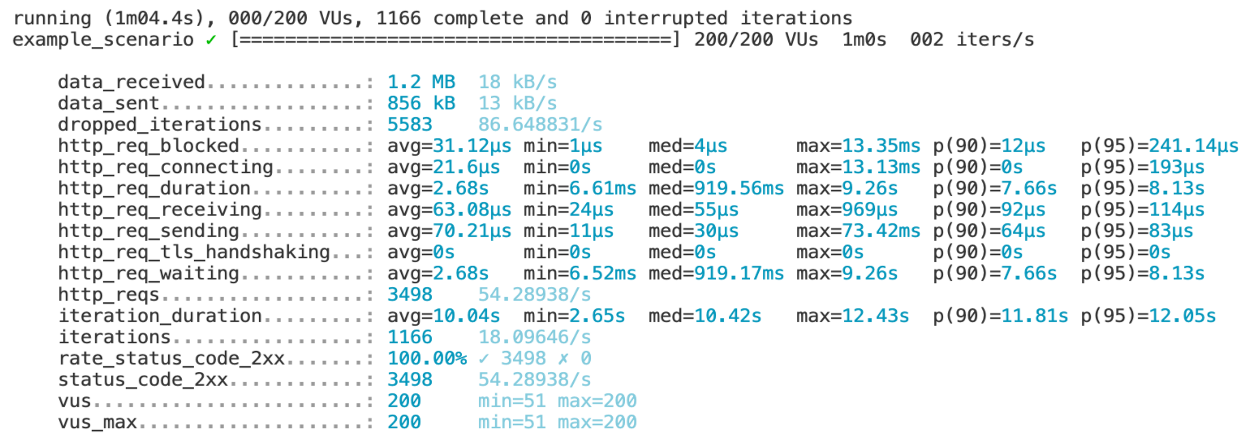


Figure 5.7.3: Heatmap and Graph (P95) of background running algorithm.

By doing this, we reduced the response duration by 27.87 seconds. For the latter, by placing the algorithm in a background process, we can significantly reduce the P(95) from 27 seconds to 8.13 seconds whereas the worst latency is down from 27.87 seconds to 9.26 seconds. In general the latency is down by 66.7 %.

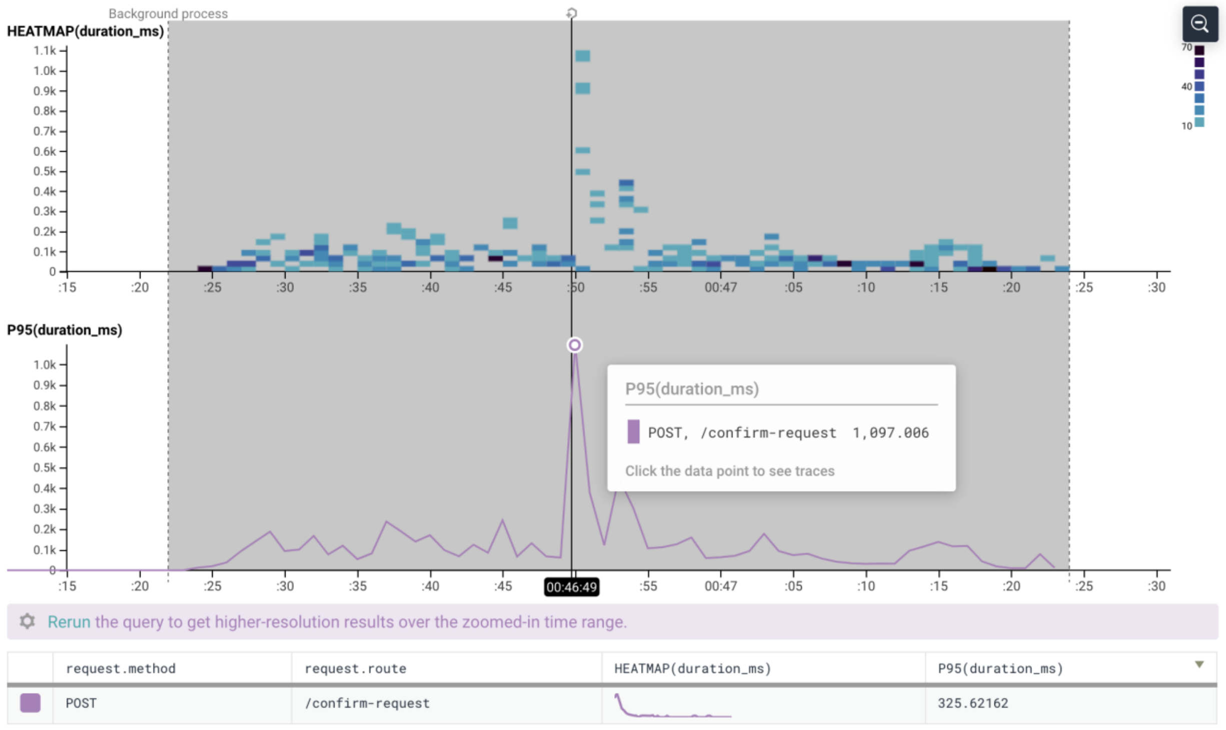


```
running (1m04.4s), 000/200 VUs, 1166 complete and 0 interrupted iterations
example_scenario ✓ [======================================] 200/200 VUs  1m0s  002 iters/s

     data_received..............: 1.2 MB  18 kB/s
     data_sent..................: 856 kB  13 kB/s
     dropped_iterations.........: 5583    86.648831/s
     http_req_blocked...........: avg=31.12µs min=1µs    med=4µs      max=13.35ms p(90)=12µs   p(95)=241.14µs
     http_req_connecting........: avg=21.6µs  min=0s     med=0s       max=13.13ms p(90)=0s     p(95)=193µs
     http_req_duration..........: avg=2.68s   min=6.61ms med=919.56ms max=9.26s   p(90)=7.66s  p(95)=8.13s
     http_req_receiving.........: avg=63.08µs min=24µs   med=55µs     max=969µs   p(90)=92µs   p(95)=114µs
     http_req_sending...........: avg=70.21µs min=11µs   med=30µs     max=73.42ms p(90)=64µs   p(95)=83µs
     http_req_tls_handshaking...: avg=0s      min=0s     med=0s       max=0s      p(90)=0s     p(95)=0s
     http_req_waiting...........: avg=2.68s   min=6.52ms med=919.17ms max=9.26s   p(90)=7.66s  p(95)=8.13s
     http_reqs..................: 3498    54.28938/s
     iteration_duration.........: avg=10.04s  min=2.65s  med=10.42s   max=12.43s  p(90)=11.81s p(95)=12.05s
     iterations.................: 1166    18.09646/s
     rate_status_code_2xx.......: 100.00% ✓ 3498 ✗ 0
     status_code_2xx............: 3498    54.28938/s
     vus........................: 200     min=51 max=200
     vus_max....................: 200     min=51 max=200
```

Figure 5.7.4: Statistics generated by K6 for the background running algorithm.

We then further optimized the matching algorithm and ‘confirm-request’ endpoint with raw SQL statement and Parallel queries. While the POST(‘confirm-request’) endpoint has 5 queries, the matching algorithm uses 20 queries. Initially we used TypeORM to query which led the longest latency to be 9.36 seconds. We optimized the endpoint by writing raw SQL statements for the 25 queries. As a result, the average P95 latency is 325.62 ms and the worst latency is 1097 ms down from 9 seconds by 88%.



P95(duration_ms)

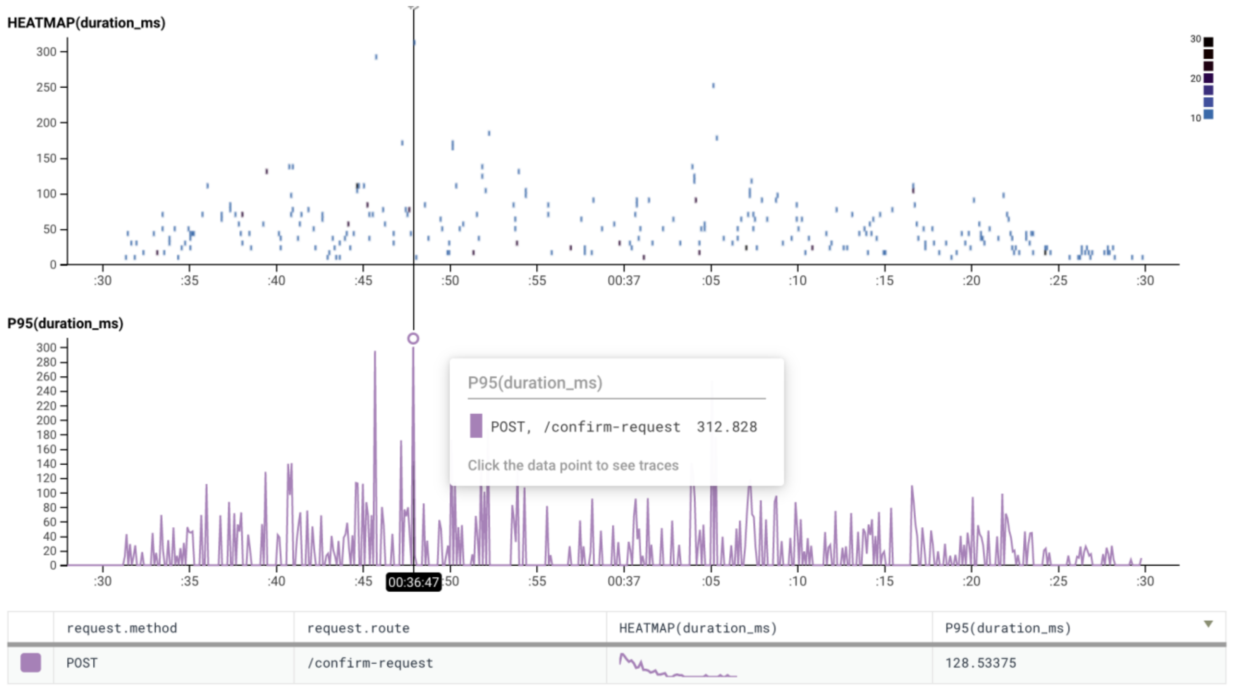
POST, /confirm-request 1,097.006

Click the data point to see traces

	request.method	request.route	HEATMAP(duration_ms)	P95(duration_ms)
🟪	POST	/confirm-request		325.62162

Figure 5.7.5: Heatmap and Graph (P95) after raw SQL statement and parallel queries of POST('confirm-request') endpoint and algorithm.
https://ui.honeycomb.io/xcurrency/datasets/bespin/result/UNT5osW3cw

Deploying the request exchanging algorithm on a separate server and making it run intervally to check for a match would significantly drop the response waiting time. The reason is the node js server is now free from performing the computational expensive algorithm, thus having much more cycles serving requests. Instead of running the algorithm by each request from clients, it is possible to set the algorithm up on an interval. For each interval, the algorithm picks a request from the database and checks for a match.



	request.method	request.route	HEATMAP(duration_ms)	P95(duration_ms)
🟪	POST	/confirm-request		128.53375

Figure 5.7.6: Heatmap and graph (P95) of POST('confirm-request') endpoint.
https://ui.honeycomb.io/xcurrency/datasets/bespin/result/kHngwpbCpDf

Performance improvements with our optimizations include the average response of p95 being only 128.53 ms and the worst case being only 312.8 ms. The response time has improved by 58% compared to placing the background process on the same node js server.

Overall, the latency of the POST('confirm-request') endpoint has dropped from

5.8 Subscription

Baseline Performance

The load testing scenario that we used was the executor - ramping arrival rate that simulated a variable number of iterations in a specific period of time. The maximum VU was 200 and it started with 50. The ramp up duration was 30s. It was important to ramp up gradually and not immediately to simulate a real world scenario. The load test was run on a Lenovo laptop with Microsoft Windows 10, Intel(R) Core(™) i5-10210U CPU @1.60GHz, and 8GB RAM.

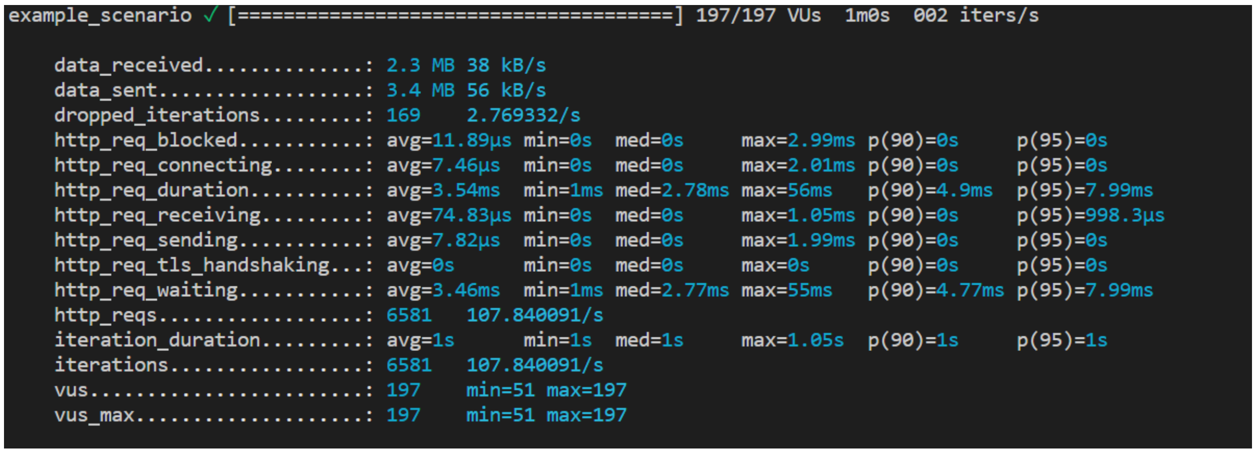
Hypothesis

By changing our Profile page and Transfer page to use subscription instead of polling, we believe we can improve the performance of our application. In theory, it is always true that polling is an inefficient solution compared to subscription because polling will perform database queries and add overhead to the application regardless of whether an update is needed; it merely runs as many times as it is set to run based on the poll interval. In contrast, the subscription only performs updates as needed due to the publish and notification process it entails, and it contains information regarding the exact account that was changed along with its new details. Thus, the Profile page and Transfers page will not arbitrarily perform extra queries with this subscription optimization, but they will still display the same, updated information, which implies that the performance is bound to increase and latency is bound to decrease here due to this changed structure of the application from polling to subscription.

<u>Implementation of Proposed Solution</u>

For this tricky optimization, we changed from using polling to subscription for our Profile and Transfers pages, meaning whenever there is a transfer between accounts, a transfer request, or a new account is created, PubSub uses publish to inform the subscription of an update, and then the respective pages subscribe to these changes to automatically update their information. Since we were forewarned to describe this optimization since it would be rather difficult to test a subscription, we did not perform load testing here after completing the optimization as instructed. Rather, we manually verify with multiple tabs open that when one of the relevant changes occurs as mentioned above, the Profile page and Transfers page do automatically display the new information without needing to refresh the page. Since we will not be able to measure the performance improvement compared to the baseline metrics, we cannot numerically quantify the percentage increase.

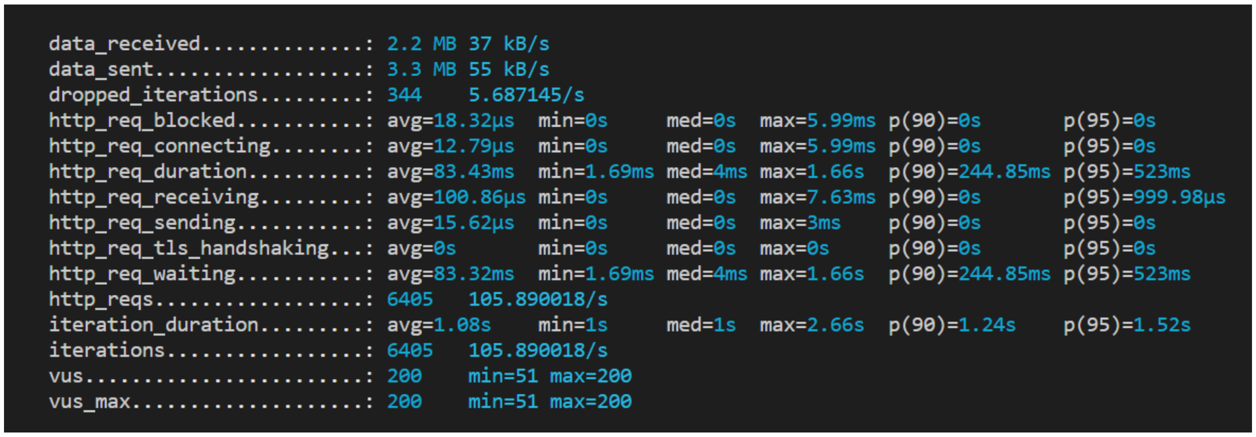
To try to test and simulate subscription, we have provided tests with polling and without polling to show how subscription could optimize our application with metrics, but of course, this is a loose estimation since we could not load test subscription alone.



```
example_scenario ✓ [======================================] 197/197 VUs  1m0s  002 iters/s

    data_received..............: 2.3 MB 38 kB/s
    data_sent..................: 3.4 MB 56 kB/s
    dropped_iterations.........: 169    2.769332/s
    http_req_blocked...........: avg=11.89µs min=0s   med=0s      max=2.99ms p(90)=0s      p(95)=0s
    http_req_connecting........: avg=7.46µs  min=0s   med=0s      max=2.01ms p(90)=0s      p(95)=0s
    http_req_duration..........: avg=3.54ms  min=1ms  med=2.78ms  max=56ms   p(90)=4.9ms   p(95)=7.99ms
    http_req_receiving.........: avg=74.83µs min=0s   med=0s      max=1.05ms p(90)=0s      p(95)=998.3µs
    http_req_sending...........: avg=7.82µs  min=0s   med=0s      max=1.99ms p(90)=0s      p(95)=0s
    http_req_tls_handshaking...: avg=0s      min=0s   med=0s      max=0s     p(90)=0s      p(95)=0s
    http_req_waiting...........: avg=3.46ms  min=1ms  med=2.77ms  max=55ms   p(90)=4.77ms  p(95)=7.99ms
    http_reqs..................: 6581   107.840091/s
    iteration_duration.........: avg=1s      min=1s   med=1s      max=1.05s  p(90)=1s      p(95)=1s
    iterations.................: 6581   107.840091/s
    vus........................: 197    min=51 max=197
    vus_max....................: 197    min=51 max=197
```

Figure 5.8.1: k6 test results using polling in the application.



```
data_received..............: 2.2 MB 37 kB/s
data_sent..................: 3.3 MB 55 kB/s
dropped_iterations.........: 344    5.687145/s
http_req_blocked...........: avg=18.32µs  min=0s     med=0s   max=5.99ms p(90)=0s       p(95)=0s
http_req_connecting........: avg=12.79µs  min=0s     med=0s   max=5.99ms p(90)=0s       p(95)=0s
http_req_duration..........: avg=83.43ms  min=1.69ms med=4ms  max=1.66s  p(90)=244.85ms p(95)=523ms
http_req_receiving.........: avg=100.86µs min=0s     med=0s   max=7.63ms p(90)=0s       p(95)=999.98µs
http_req_sending...........: avg=15.62µs  min=0s     med=0s   max=3ms    p(90)=0s       p(95)=0s
http_req_tls_handshaking...: avg=0s       min=0s     med=0s   max=0s     p(90)=0s       p(95)=0s
http_req_waiting...........: avg=83.32ms  min=1.69ms med=4ms  max=1.66s  p(90)=244.85ms p(95)=523ms
http_reqs..................: 6405   105.890018/s
iteration_duration.........: avg=1.08s    min=1s     med=1s   max=2.66s  p(90)=1.24s    p(95)=1.52s
iterations.................: 6405   105.890018/s
vus........................: 200    min=51 max=200
vus_max....................: 200    min=51 max=200
```

Figure 5.8.2: k6 test results without using polling in the application, this is supposed to give a clearer view of the subscription.

Considering the data obtained for this optimization, we expected our results to include the subscription showing better performance than polling due to the nature of this optimization as discussed. From the analysis and figures above, it seems as if running the application with and without polling yielded almost identical results, with no improvement or deprovement from the baseline performance, which doesn't exactly follow our expectations. We believed that the tests without polling would show higher performance, but that does not seem like the case. Regardless, since in theory subscription is more efficient and optimal compared to polling, we believe that this optimization should improve our application's performance. From our manual verification about the functionality of subscription working as expected, we do not believe this fix has created any new problems, but it is worth noting that making the same optimization from polling to subscription would also prove beneficial for our Transfers page.

5.9 Vertical Scaling in AWS

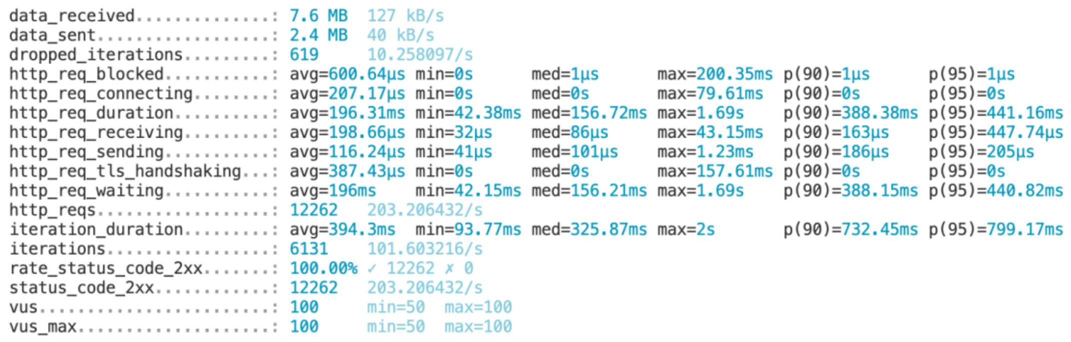
Baseline performance:

We deployed our app on ECS using terraform with the following configurations, which correspond to the t2.small ECS instance type:

- 1 instance of machine to run the app
- 1024 CPU units = 1 vCPU
- 2048 Mib of RAM

We load tested our app using K6 with the same ramping arrival rate executor scenario used throughout this report, with a ramp up to 200 concurrent users and a ramping down to zero users.

We tested the basic functionality of our app: signup and log in processes.



```
data_received..............: 7.6 MB  127 kB/s
data_sent..................: 2.4 MB  40 kB/s
dropped_iterations.........: 619     10.258097/s
http_req_blocked...........: avg=600.64µs min=0s      med=1µs      max=200.35ms p(90)=1µs      p(95)=1µs
http_req_connecting........: avg=207.17µs min=0s      med=0s       max=79.61ms  p(90)=0s       p(95)=0s
http_req_duration..........: avg=196.31ms min=42.38ms med=156.72ms max=1.69s    p(90)=388.38ms p(95)=441.16ms
http_req_receiving.........: avg=198.66µs min=32µs    med=86µs     max=43.15ms  p(90)=163µs    p(95)=447.74µs
http_req_sending...........: avg=116.24µs min=41µs    med=101µs    max=1.23ms   p(90)=186µs    p(95)=205µs
http_req_tls_handshaking...: avg=387.43µs min=0s      med=0s       max=157.61ms p(90)=0s       p(95)=0s
http_req_waiting...........: avg=196ms    min=42.15ms med=156.21ms max=1.69s    p(90)=388.15ms p(95)=440.82ms
http_reqs..................: 12262   203.206432/s
iteration_duration.........: avg=394.3ms  min=93.77ms med=325.87ms max=2s       p(90)=732.45ms p(95)=799.17ms
iterations.................: 6131    101.603216/s
rate_status_code_2xx.......: 100.00% ✓ 12262 ✗ 0
status_code_2xx............: 12262   203.206432/s
vus........................: 100     min=50  max=100
vus_max....................: 100     min=50  max=100
```

Figure 6.4.1 Statistics from k6 for the load testing

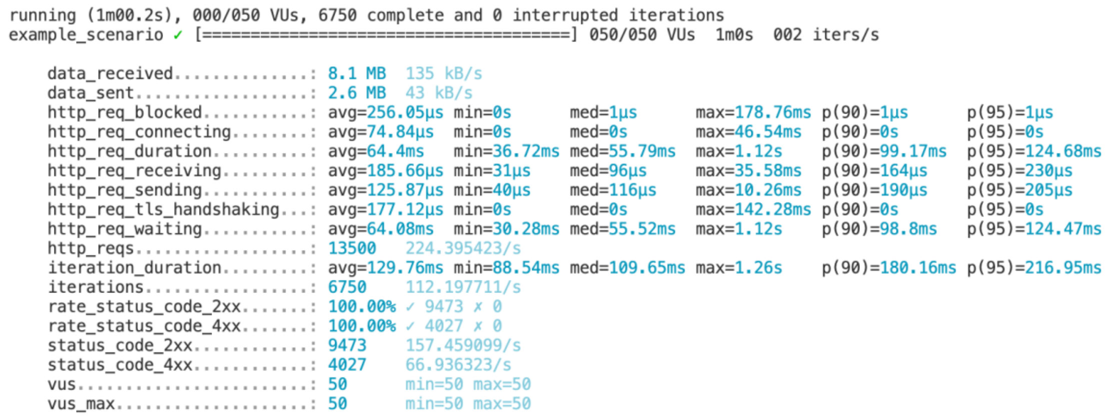
From the log output, the maximum response latency is 1.69 seconds and the median is 156.21 ms whereas the P(95) average is 440.82 seconds. These response times are acceptable for our testing scenario.

Hypothesis

______Considering the many different machine types available for deploying applications, we predicted that finding a better suited one would undoubtedly improve our application performance. Also, since the machine we can use once we deploy to AWS is much better than our locan one, we predict significant performance improvements. After doing some research, we ended up predicting that the AWS t2.small machine would be a suitable option for our application.

Implementation of Proposed Solution

We can scale the web app horizontally by increasing the number of instances to run the Docker contains. Since Node.Js is single threaded, scaling the VCpu unit would not help the performance. Furthermore, since 2 Gb is enough for an app, we did not increase the memory either.



```
running (1m00.2s), 000/050 VUs, 6750 complete and 0 interrupted iterations
example_scenario ✓ [======================================] 050/050 VUs  1m0s  002 iters/s

    data_received..............: 8.1 MB  135 kB/s
    data_sent..................: 2.6 MB  43 kB/s
    http_req_blocked...........: avg=256.05µs min=0s      med=1µs      max=178.76ms p(90)=1µs      p(95)=1µs
    http_req_connecting........: avg=74.84µs  min=0s      med=0s       max=46.54ms  p(90)=0s       p(95)=0s
    http_req_duration..........: avg=64.4ms   min=36.72ms med=55.79ms  max=1.12s    p(90)=99.17ms  p(95)=124.68ms
    http_req_receiving.........: avg=185.66µs min=31µs    med=96µs     max=35.58ms  p(90)=164µs    p(95)=230µs
    http_req_sending...........: avg=125.87µs min=40µs    med=116µs    max=10.26ms  p(90)=190µs    p(95)=205µs
    http_req_tls_handshaking...: avg=177.12µs min=0s      med=0s       max=142.28ms p(90)=0s       p(95)=0s
    http_req_waiting...........: avg=64.08ms  min=30.28ms med=55.52ms  max=1.12s    p(90)=98.8ms   p(95)=124.47ms
    http_reqs..................: 13500   224.395423/s
    iteration_duration.........: avg=129.76ms min=88.54ms med=109.65ms max=1.26s    p(90)=180.16ms p(95)=216.95ms
    iterations.................: 6750    112.197711/s
    rate_status_code_2xx.......: 100.00% ✓ 9473 ✗ 0
    rate_status_code_4xx.......: 100.00% ✓ 4027 ✗ 0
    status_code_2xx............: 9473    157.459099/s
    status_code_4xx............: 4027    66.936323/s
    vus........................: 50      min=50 max=50
    vus_max....................: 50      min=50 max=50
```

Figure 6.4.2 Statistics from k6 for the improved load testing

From the statistics, it is clear the median response latency is only 55.52 ms down by 65%. The worst waiting time is 1.12 seconds down from 1.69 seconds. Furthermore, the P(95) has significantly improved from 440.82 ms to 124.47 ms down by 72%.

As evidenced, the app performance has significantly improved by vertical scaling - adding more instances to run the docker containers because there are 5 more Node js servers running to handle the client's requests. All of the requests are forwarded by load balancer to spread the tasks evenly to the 6 servers.

6. Future Improvements

6.1 Improve Request Addition, Execution, and Deletion Time Complexity

Dictionaries, binary-trees, and linked-lists could be used to improve request creation, request deletion, and request execution's time complexity and to prioritize requests that have better rates and arrive early.

If there is a currency pair X, Y, there would be variables that indicates the initial requestId that has the best rate from X to Y and from Y to X, binary-trees that contain the distinct rates from X to Y and from Y to X, and linked-lists that branch from the binary tree nodes. The linked-list nodes would contain the specific request details. The dictionaries would map from the requestId to the linked-list node.

The aforementioned data structure would result in O(log K) request deletion, O(log K) request execution, and O(log K) request addition, where K is the number of distinct rates. However, the average time complexity would be O(1) for these operations because it is likely that there will be multiple orders at the same distinct rate.

6.2 Improve Match Probability

There is no guarantee that requests from X to Y are sufficient to match requests from Y to X, and, therefore, matches could be rare. A breadth-first search approach could improve match probability. If there are requests from A to B, from B to C, and from C to A, these requests could be matched. This process could be extrapolated to include 180 currencies.