

Statsprime

A stats tracker for League of Legends

Team: twitchprime

Xiaojian Chen

Weizhe Li

Xiaolei Luo

Weisheng Zhang

Table of Contents

1	Introduction	3
2	Baseline Performance	7
	2.1 Testing Tool -- K6	7
	2.2 User Behavior Simulation Load Test	7
	2.3 K6 Test Scenario	8
	2.4 Testing Environment	8
	2.5 Test Results	8
	2.5.1 "Player-detail" Page	8
	2.5.2 "Match-detail" Page	10
3	Hypothesis	10
4	Experiments & Improvements	11
	4.1 Redesigning MySQL Searching Logic and Implementation	11
	4.1.1 Fix Incorrect "Eager" Relation	11
	4.1.2 Integrating(Reducing) SQL Table	13
	4.1.3 Redesign Database Searching Logic	14
	4.1.3.1 Redesign Database Searching Logic of Rendering "Player-detail"	15
	4.1.3.2 Redesign Database Searching Logic of Rendering "Match-detail"	17
	4.2 Server-side Caching	20
	4.2.1 Caching Mechanism Design	20
	4.2.2 Performance (After Caching)	22
	4.2.3 User Behaviour Simulation Load Test	24
5	Vertical & Horizontal Scalings	25
	5.1 Scaling Tool & Settings	25
	5.2 Problems We Ran Into	26
	5.3 Vertical Scaling	27
	5.4 Horizontal Scaling	28
6	Summary & Conclusions	30
	6.1 Further Development	30
	6.1.1 Caching the "Search-Name to Index" Map	30
	6.2 Conclusion	30

1. Introduction

Statsprime is a web application for MOBA game League of Legends, developed by Riot Games. Statsprime enables anyone with a Riot account, most likely a League of Legends player, to view a player's summoner information and history of the 10 most recent matches by entering any summoner name and Riot API key. Since a development key cannot be used in any publicly available product, Riot API key input is anonymized. Once a key is inserted, only the summoner name is needed for any search. Upon finding a player, the player detail page displays critical information such as summoner level, tier/rank, win rate and recent match date/game type. Upon clicking any of the match history rows, users have access to detailed match history featuring champion selection, champion level, item choices, Kill/Death/Assist (K/D/A), gold earned) for every player in the match. Every row has an independent collapse that shows a player's rune choice. Users can click any champion icon in any match detail page to view a player's detail where the player played that champion in that specific match.

The following screenshots show the pipeline of how to get detail of a specific match played by a specific player.

LEAGUE OF LEGENDS STATS TRACKER

Riot API Key


Player Name


@twitchprime

Figure 1: default home page of the application

Insert Riot API Key in the first textbox. Since Riot API Key should not be publicly exposed, it is anonymized (as mentioned above) like passwords. Press the blue button next to it to store temporarily on the server.

Insert summoner name in the second textbox. Press the blue button next to it to start a search.


Peng Yiliang


MASTER I 29 LP

WINS: 59
LOSSES: 29
WIN RATE: 67.05%
SUMMONER LEVEL: 51








	GAME MODE	GAME DATE
	Summoner's Rift Ranked Solo	12/7/2020
	Summoner's Rift Ranked Solo	12/7/2020
	Summoner's Rift Ranked Solo	12/7/2020
	Summoner's Rift Ranked Solo	12/7/2020
	Summoner's Rift Ranked Solo	12/3/2020
	Summoner's Rift Ranked Solo	12/3/2020
	Summoner's Rift Ranked Solo	12/3/2020

Figure 2: an example of the player detail page

This is a screenshot of a sample search. It should give a clear idea of how the player detail page looks like. Summoner name and profile icon are displayed at the top, followed by tier icon, tier, rank, and league point information. Wins, Losses, win rate, and summoner level are displayed as tag-like objects. The main body of the player detail page is a table of 10 most recent matches played with champion icon, game mode, and game date information available.

Game duration: 19 min 32 s
VICTORY (Red Team)




















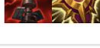






























					Level	K/D/A	Damage	CS	Gold	
			PENG YILIANG	  	12	5/0/4	5645	22	8324	^
										
			CUPCAKES29	     	13	6/2/3	8315	147	8147	v
			PERMAROAM908	     	9	3/3/10	4963	23	6035	v
			THOMAS YU	     	11	4/1/5	8810	155	8748	v
			C9 FUDGEY	     	14	5/0/0	11797	177	8728	v
										
DEFEAT (Blue Team)										
			XSTOLEN	  	9	0/8/3	7208	96	5091	v

Figure 3: an example of the match detail page

This is a screenshot of a sample match. It should give a clear idea of how the match detail page looks like. Columns are displayed as follows. Champion choice, summoner spell choice, primary/secondary rune icon, summoner name, item choice, champion level, kill/death/assist (K/D/A), damage dealt with champions, creeps slain, and gold earned. Below the main row is the detailed rune choice (perks).

In terms of where our data originate from, image data is downloaded from Data Dragon, a Riot API library. It is stored in assets as part of the server. Text data is acquired by making queries to Riot Games API. One must go through the following steps to fetch player/match detail.

- Use `/lol/summer/v4/summoners/by-name/{summonerName}` to get accountId, profileId, summonerLevel
- Use `/lol/league/v4/entries/by-summoner/{encryptedSummonerId}` to get all league entries (tier, rank, wins, losses) in all queues (ranked solo, ranked flex, etc.) for a given summonerId.
- Use `/lol/match/v4/matchlists/by-account/{encryptedAccountId}` to get a list of matchIds on given accountId
- Use `/lol/match/v4/matches/{matchId}` to get detail on a certain match (champion selection, item choices, K/D/A, etc)

Since this application is not developed for production purposes, a development key from Riot Games is the only available option. However, a development key comes with rate limits. According to Riot Developer Portal documentation, a development key can make 20 requests every second/ 100 requests every 2 minutes [1]. This limits the server's ability to process requests on different summoner names, but it is decided that more fetching power can be expanded once other aspects of the server are set. To accommodate this limitation, the server only updates player or match details if they are older than one hour. Given an average game length of 35 - 40 minutes per game (including queuing, draft phase, and actual game time), this update scheme is a reasonable approximation since 1-2 games will conclude during a one-hour interval, most likely. Additionally, player detail will be stored to Redis if it is fetched 10 times or more. The performance boost that these schemes bring about will be discussed in later sections of the report.

Statsprime has four entities. The following diagrams show dependency relations among them, including one-to-one and one-to-many relations. There is one interesting yet tricky thing about these entities. Both summoner and match participant represent certain players, but they are conceptually different as a summoner can be one of the ten-match participants in many matches.

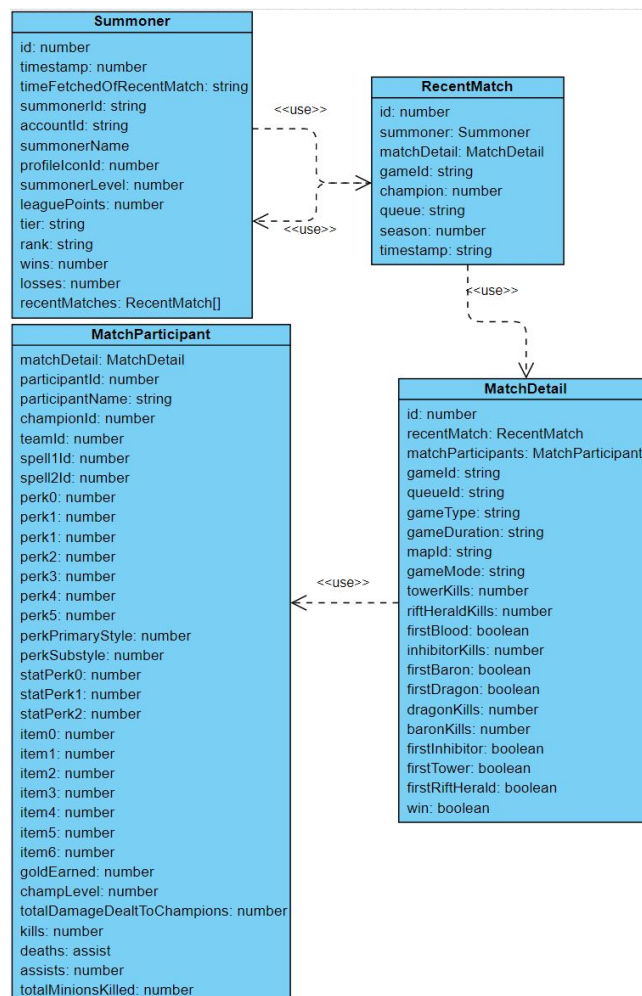


Figure 4: different entities used and the relations between them

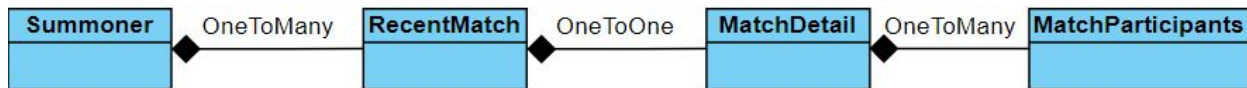


Figure 5: more relations between entities

Note that Riot API does return role and lane information in `/lol/match/v4/matchlists/by-account/{encryptedAccountId}`. However, multiple champions in the game can flex among multiple roles. This information from API is not always accurate and therefore it is decided that role and lane is not included in the match history table. Another note: both teams, blue and red, have the following attributes: towerKills, riftHeraldKills, firstBlood, inhibitorKills, firstBaron, firstDragon, dragonKills, baronKills, firstInhibitor, firstTower, firstRiftHerald, win.

2. Baseline Performance

2.1 Testing Tool -- K6

Because our application's performance can be easily tested by simulating the user behavior of accessing certain HTTP endpoints, so K6 is the best tool for us to use to test our application's performance. And we will observe the performance on the K6 test result and Honeycomb.

2.2 User Behaviour Simulation Load Test

Ideally, the critical user path can be: First, the user searching a summoner's game stat and recent 10 matches result by accessing "**`app/player-detail/[SummonerName]`**". And then the user clicks on one of the matches to see the match detail result, the HTTP endpoint will be "**`app/match-detail/[matchID]`**".

However, the process of the "**`player-detail`**" page is much more complex than the process of the "**`match-detail`**" page, and it invoked more MySQL database access than the "**`match-detail`**" page.

Thus, we decided to get baseline performance and improve them separately.

```
export default function () {  
  http.get('http://localhost:3000/app/player-detail/[SummonerName]')  
}
```

The function above will simulate the first part of the critical user path, which is searching a summoner's game stat and recent 10 matches results.

```
export default function () {
  http.get('http://localhost:3000/app/match-detail/[MatchId]')
}
```

The function above will simulate the second part of the critical user path, which is looking up the match detail result.

We will test the full critical user path after we can obtain some good performance on both parts.

2.3 K6 Test Scenario

In our K6 testsript, we are mainly using the executor 'ramping-arrival-rate' to simulate the case that there are more and more requests are sending to our server concurrently. To get the baseline performance, we set the stages to be: first, ramp up to 1 iter/sec in 10s and stay on it for the 20s. Next, ramp up to 2 iter/sec in 10s and stay on it for the 20s. Then, ramp up to 3 iter/sec in 10s and stay on it for the 20s. Finally, ramp down to 0 iter/sec in the 30s.

2.4 Testing Environment

All tests are conducted on a 2020 13-inch MacBook Pro with 2 GHz Quad-Core i5 and 16 GB 3733 MHz LPDDR4X.

2.5 Test Results

2.5.1 "Player-detail" Page

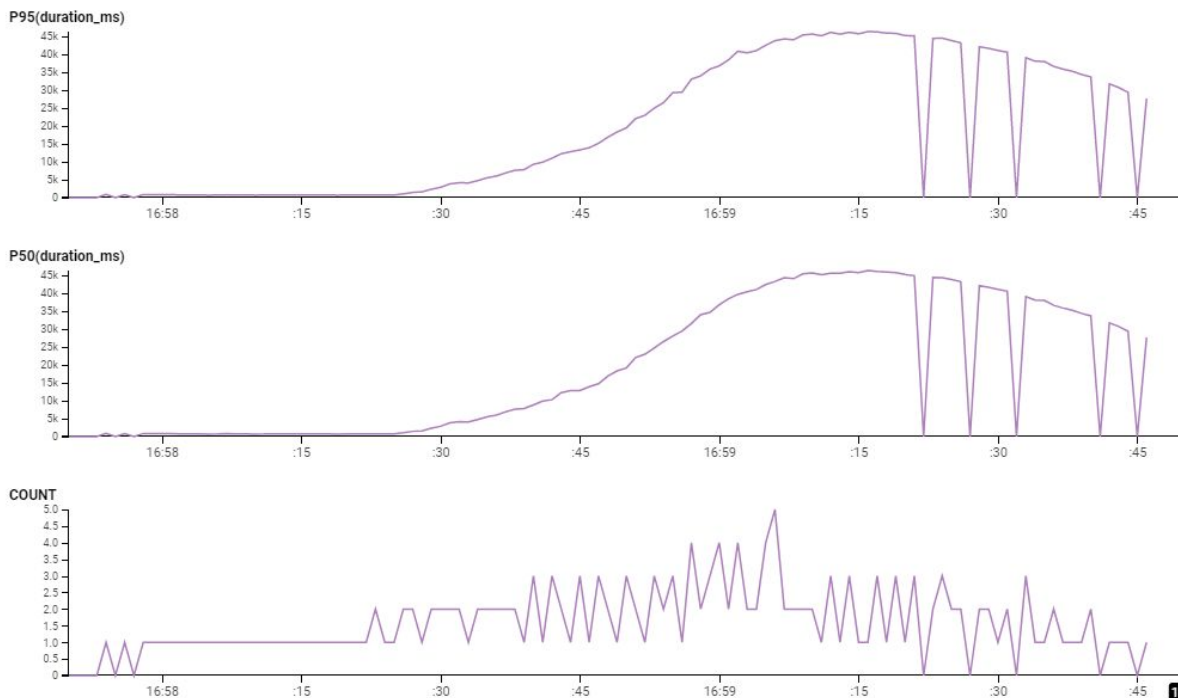


Figure 6: Honeycomb Result of Player Detail Request(Baseline performance)

By running the test of accessing the “player-detail”. We got the result graphs above from the HoneyComb. As we can see the P50 and P95 graphs are identical, which means no outlier affects our test result.

When the event/sec is 1, the duration is a very low value of less than 1s, which is great. But when the event/sec increases to 2, the duration is increased to 5 to 10s. As the event/sec increases, the duration increases more rapidly, and it reaches the maximum value of 45s when event/sec equal to 5. Thus, we can conclude that **the “player-detail” page’s QPS before performance degrades is 1.**

There is an interesting thing we observed from the count graph. Given that the count is the number of completed events per second, ideally, every time, the count should increase to a new value in 10s and stay on it for the 20s, But, after time 20s, the k6 iters/sec starts to increase from 1, but the count instead of stating on one value for 20s, it is bouncing up and down. We guess it is because our application was not fast enough to handle more than 1 quest concurrently, thus some requests are delayed and are completed along with requests of later iterations. That is why we saw the maximum count 5, while the maximum iter/sec is 3 in our k6 test script.

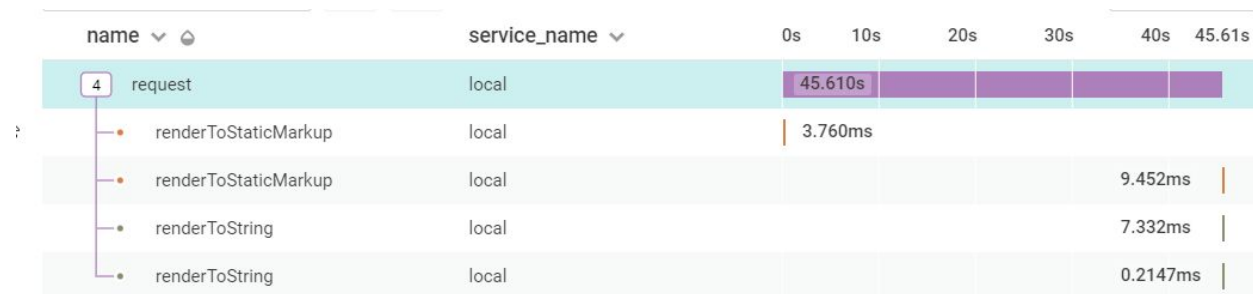


Figure 7: Honeycomb Trace Result of Player Detail Request

The graphs about are the trace result that has the highest duration time. As we can see, Most of the time is consumed by waiting for the request to be complete, the rendering only took a few ms. But we don’t know which part of the request is delaying the entire request. So for the next step, we plan to go over our code that handles the request and generate the response and examine or log messages to see where the problems at.

2.5.2 “Match-detail” Page

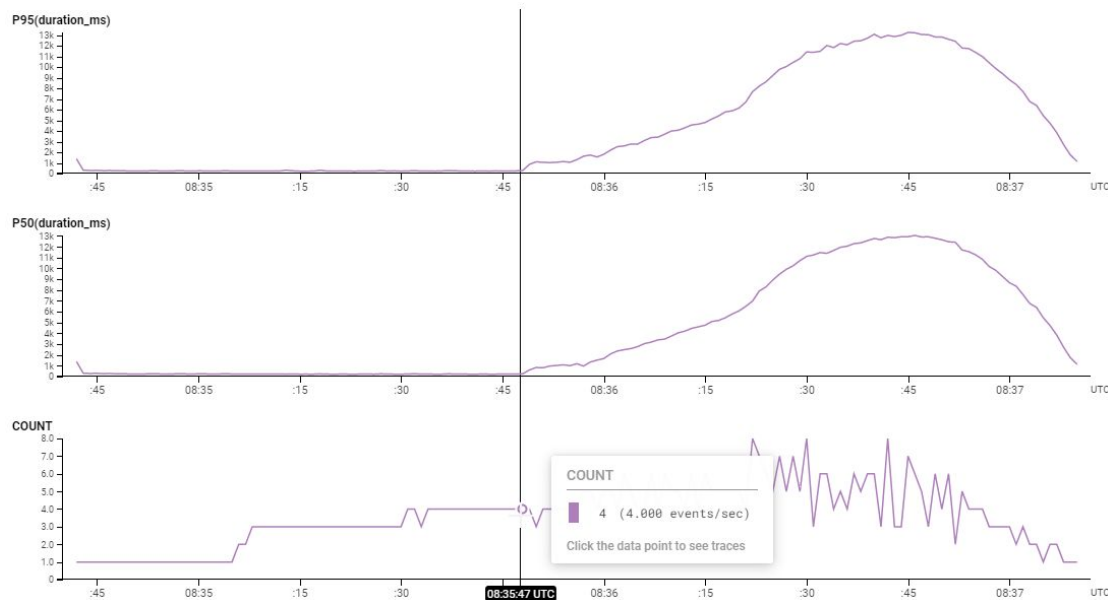


Figure 8: Honeycomb Result of Match Detail Request(Baseline performance)

By running the test of accessing the “match-detail”. We got the result graphs above from the HoneyComb. Similar to the result of accessing the “player-detail”, there is no sign of an outlier that affects our test result. However, it's performance is slightly better than “player-detail”'s. When the iters/sec is less than or equal to 4, duration remains on a very low value of less than 1s. Once the iters/sec reaches 5, the performance degrade happens. Thus, we can conclude that the “match-detail” page's QPS before performance degrades is 4.

3. Hypotheses

After examining the initial result and our code, we think most of the latency comes from the MySQL accessing process of updating data and getting data, rather than the web page rendering.

To prove our hypothesis, we have to know how much the latency effect the database accessing process has, so we comment out all the MySQL accessing process codes and let the function return a constant JSON object for page rendering. By running the same load test script, the result shows that the duration time is less than 50 ms for all the time when the iter/sec is less than 4. Thus It proves that most of the latency comes from the database accessing processes for both “player_detail” and “matchdetail”.

To explore which part of the database accessing process is slowing down our event, we turned on the SQL log function to examine the log message.

We found that for loading a pre-searched summoner's (no need to fetch the data from riot server and update our database) play-detail page, there are about 20 SQL queries invoked, which is too much. Because, ideally, given that we have set up the join relation, loading up the player stat and recent matches should only require 2 SQL queries, one for finding the table index, one for getting the actual row along with all joined recentMatches.

For loading a pre-searched summoner's match-detail page, there are 4 SQL queries, which means there is an extra removable find process in the code.

The log SQL log messages show that our database searching logic and implementation of getting "player-detail" were very poor. Thus our main goal in the improvement is reducing SQL query as much as possible. This can be done by **redesigning our database logic procedure and implementation** and **using server-side caching**.

4. Experiments & Improvements

4.1 Redesigning MySQL Searching Logic and Implementation

4.1.1 Fix Incorrect "Eager" Relation

When we examine the SQL log message of accessing the "player-detail" page, the first thing we found out is that every query of the RecentMatch entry is followed by a query of MatchDetail, which is unnecessary. A user searching their game stat and their recent matches don't mean they must look up the match detail. And this process introduces many overheads: loading up a "player-detail" page will load up 10 recent matches, and for each recent match query, there is one unnecessary query of MatchDetail. So in each k6 test iteration, there are 10 extra queries.

```
@Entity()
export class RecentMatch extends BaseEntity {
  @OneToOne(() => MatchDetail, matchDetail => matchDetail.recentMatch, {eager: true})
  matchDetail: MatchDetail
}
```

By examining our code, we found that we have incorrectly set the "eager" of the One-To-One relation from the entry RecentMatch to MatchDetail to true. It means, when any entry row of RecentMatch is loaded, the corresponding matchDetail (no matter if it exists or not) will be loaded as well.

To fix this problem, we set the "eager" to false and then run the same k6 load test again. Here is the result from Honeycomb.

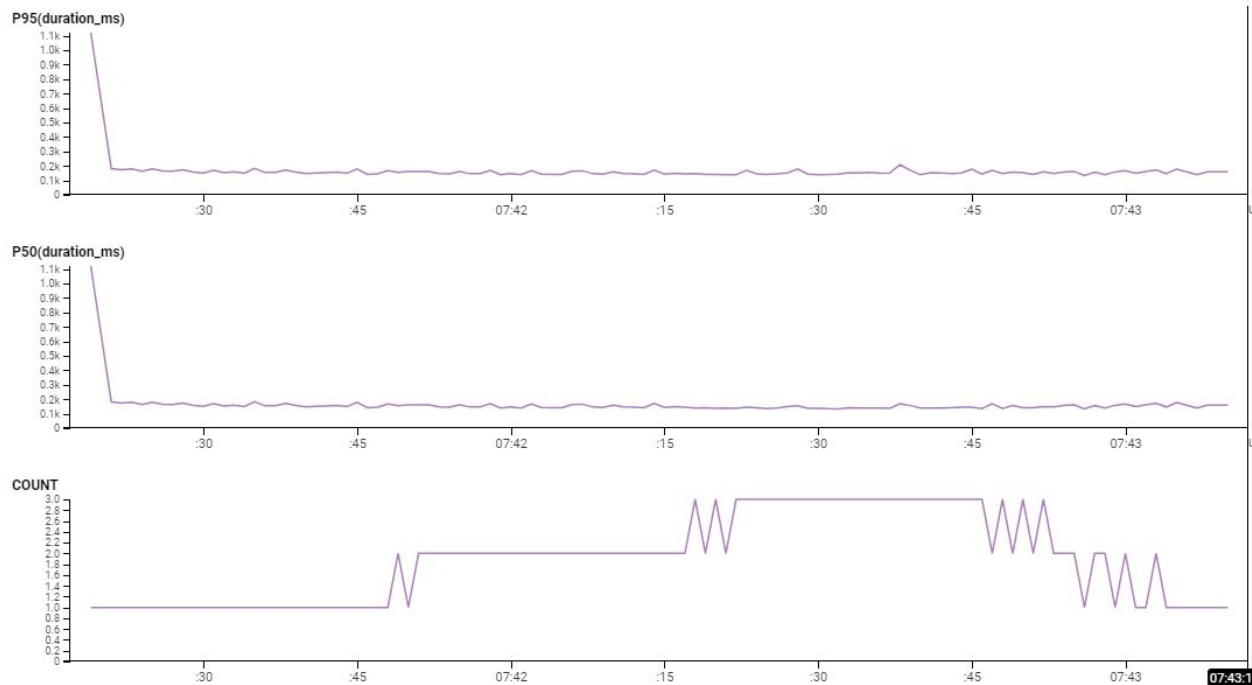


Figure 9: Honeycomb Result of Player Detail Request

As we can see from the above, in the beginning, the database has no server data yet, it took an extra time to fetch summoner's game stat and 10 recent matches data from the Riot server, thus there was a spike on the p50 and p95 graph in the beginning. After that, the application will use that data as a search result for the next 30min before they become outdated. Other than the first request, P50 and P95 are lower than 280ms all the time, which means there are no performance downgrades and QPS ≥ 3 .

To find out the new QPS before performance downgrade, we change our "ramping-arrival-rate" testscript to ramp up in 10s and stay on its 20s with the target **1, 5, 7, 8, and 9 iters/sec**. The Honeycomb result of the new load test is shown below:

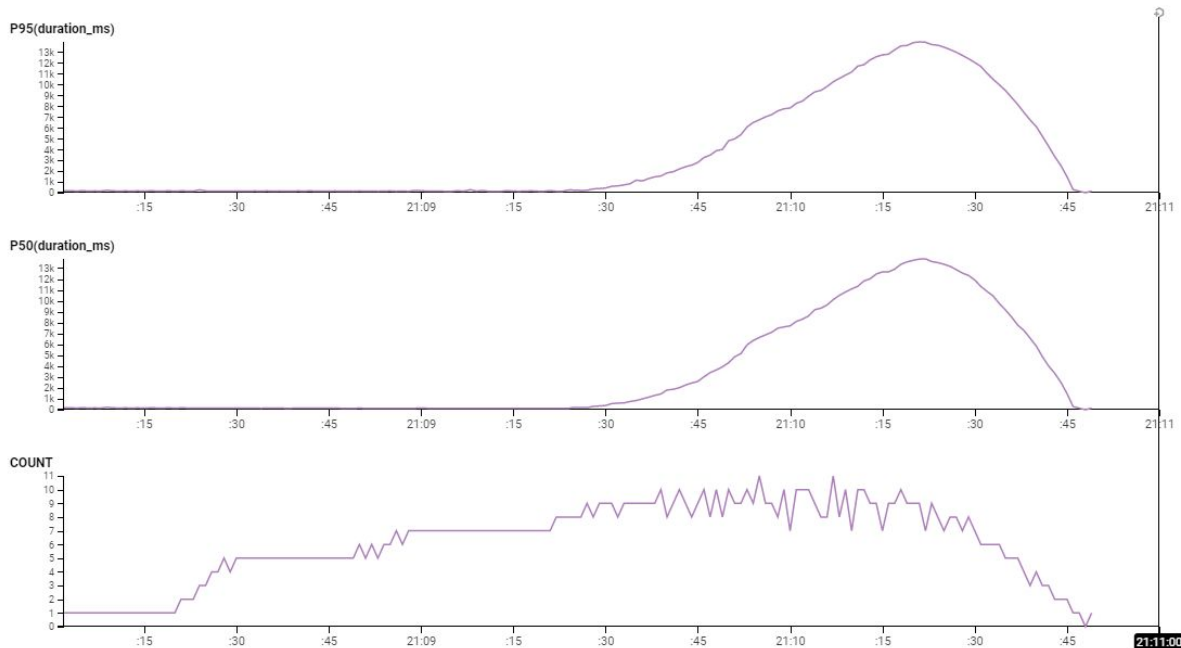


Figure 10: Honeycomb Result of Player Detail Request

As we can see from the graph above, before the iter/sec increase to 8, the p50, p95 duration time is in a low value of less than 1s. Once the iter/sec larger than or equal to 8, the performance becomes unstable, the count is bouncing up and down and the P50, P95 duration is increasing.

Thus, we can conclude that by removing the incorrect “eager”, the **“player_detail” page’s QPS before performance downgrade becomes 7**, which is **7 times better** than the initial performance.

4.1.2 Integrating(Reducing) SQL Table

In our initial implementation, there was a SQL Table called “FetchTime”, it is used to store the timestamp of the last time the application fetches/updates the summoner’s RecentMatches from the Riot server. So that the application won’t have to update the summoner’s RecentMatch table when every time the user requests the “player-detail” page unless the timestamp has passed a certain threshold.

```
@Entity()
export class FetchTime extends BaseEntity {
  @PrimaryGeneratedColumn()
  id: number

  @Column()
  summonerName: string

  @Column()
  recentMatchesTimeFetched: string
}
```

However, this FetchTime introduces two more database queries when a user requests RecentMatches or the application updating the RecentMatches. To reduce the number of database queries that would happen when users accessing the “player-detail” page, we integrated the recentAMtchesTimeFetched column into Summoner’s table.

To find out the new QPS before performance downgrade, we change our “ramping-arrival-rate” testscript to ramp up in 5s and stay on its 20s with the target **1, 10, 12, 13, 14, and 15 iters/sec**. The Honeycomb result of the new load test is shown below:

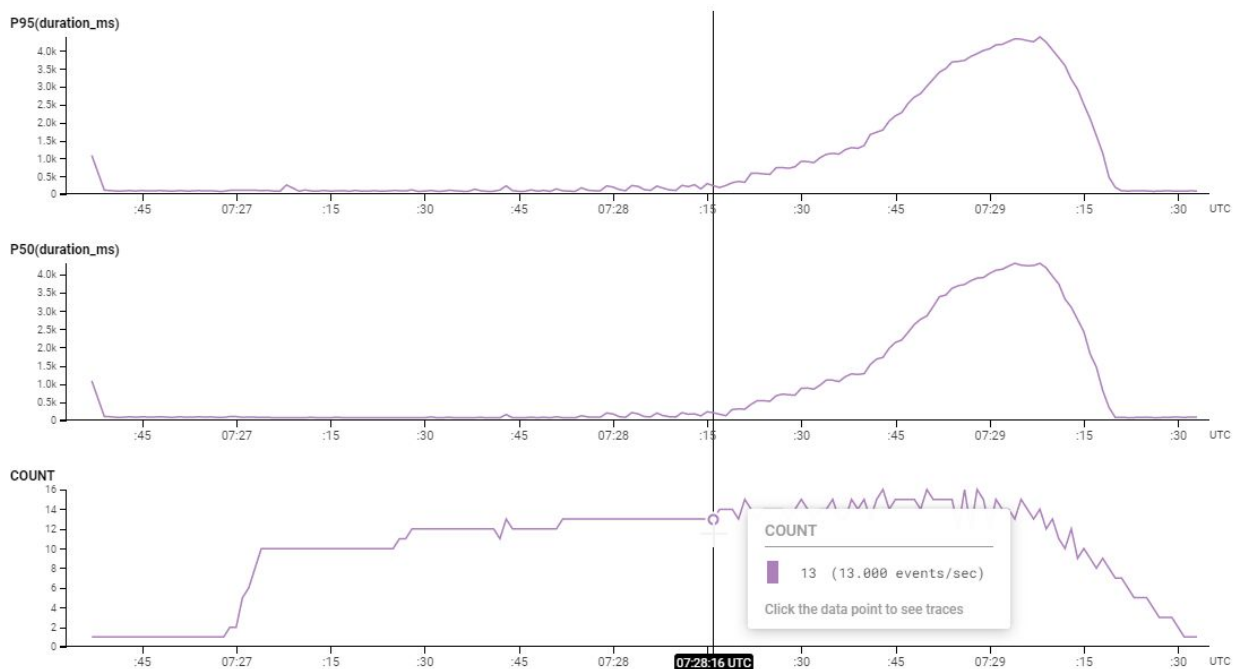


Figure 11: Honeycomb Result of Player Detail Request

Before the iter/sec increases to 13, the p50, p95 duration time is in a low value of less than 1s. Once the iter/sec larger than or equal to 13, the performance becomes unstable, the count is bouncing up and down and the P50, P95 duration is increasing.

Thus, we can conclude that by integrating the FetchTime table into the Summoner Table, **the “player_detail” page’s QPS before performance downgrade increases to 13**, which is about **2 times better** than the previous improvement’s performance.

4.1.3. Redesign Database Searching Logic

As we mentioned in the hypotheses part, for both getting the “player-detail” page and the “match-detail” page, there are many unnecessary SQL queries inside our searching process. So we redesigned the searching logic for both pages to reduce the number of SQL queries per request to 2.

4.1.3.1 Redesign Database Searching Logic of Rendering “Player-detail”

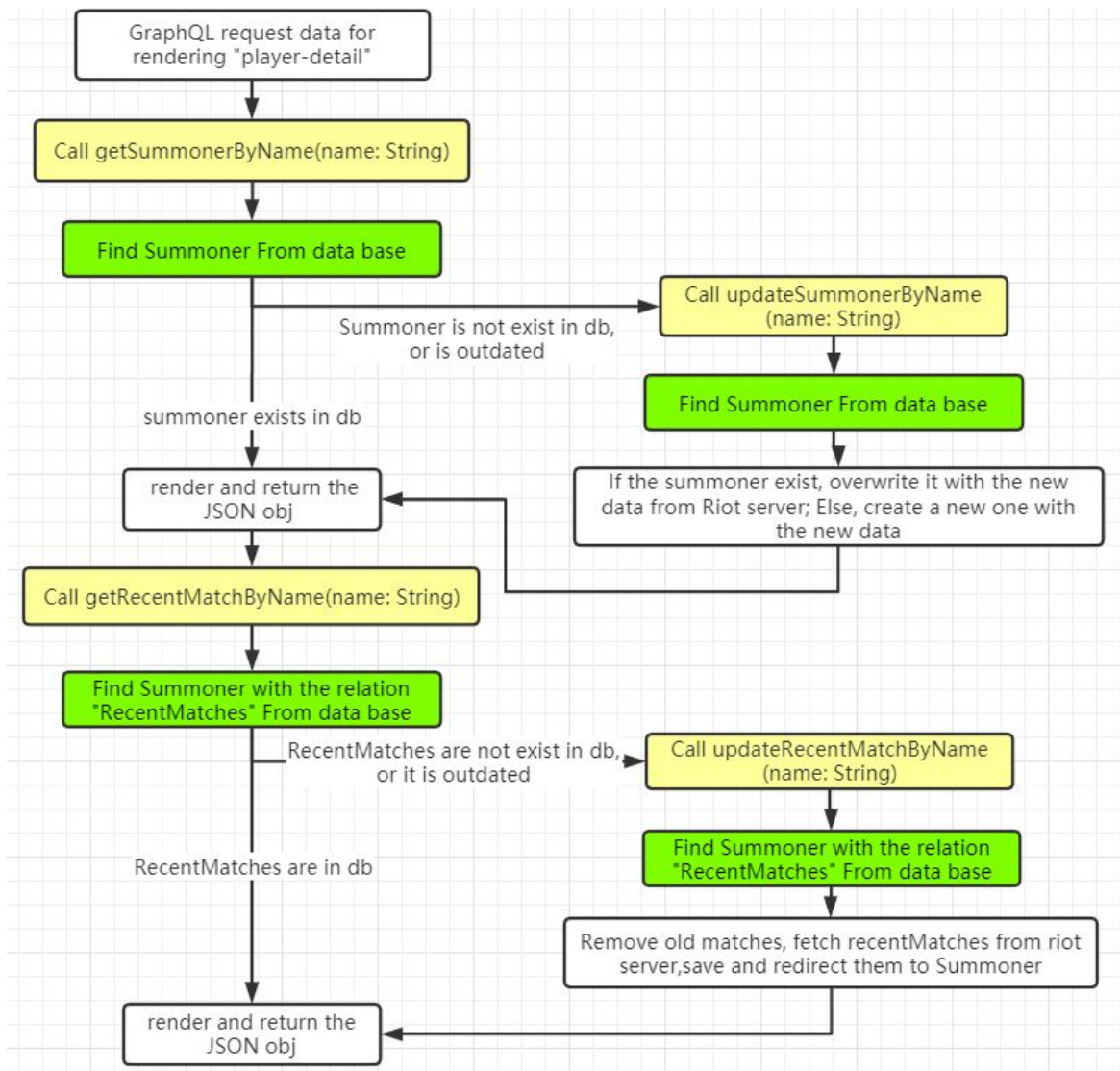


Figure 12: Initial searching procedure for rendering the “player-detail” page

The flowchart above is our initial searching procedure for rendering the “player-detail” page. When all the required data (Summoner and RecentMatch) are present and valid(not outdated), it requires 2 SQL Finds, which is 4 SQL queries (get the row index, and get the row). If the data does not exist or needs updating, it requires 4 SQL Find, which is 8 SQL queries.

By examining the codes, we discovered that 3 of these 4 SQL Finds can be eliminated. The new flowchart of the searching procedure for rendering the “player-detail” page is shown below.

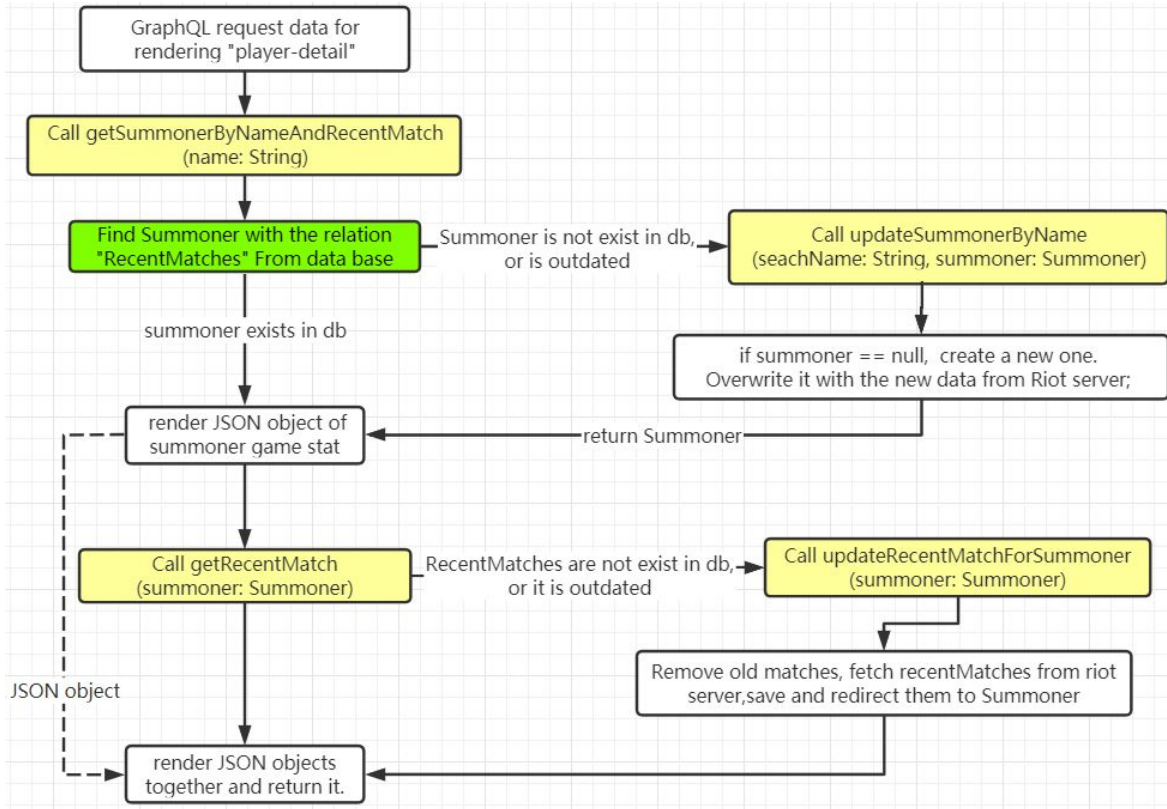


Figure 13: New searching procedure for rendering the “player-detail” page

We change or add a parameter “Summoner” to these functions so that we can do only one SQL Find on Summoner with the relation “RecentMatches” in the beginning, and then use the found summoner for the entire process of rewriting it or getting data from it. Thus, the application only requires (1 Find) **2 SQL queries** for getting a summoner’s game stat and recent 10 matches.

To find out the new QPS before performance downgrade after the improvement, we change our “ramping-arrival-rate” testscript to ramp up in 10s and stay on its 20s with the target **1, 15, 20, 22, 24, and 26 iters/sec**. The Honeycomb result of the new load test is shown below:

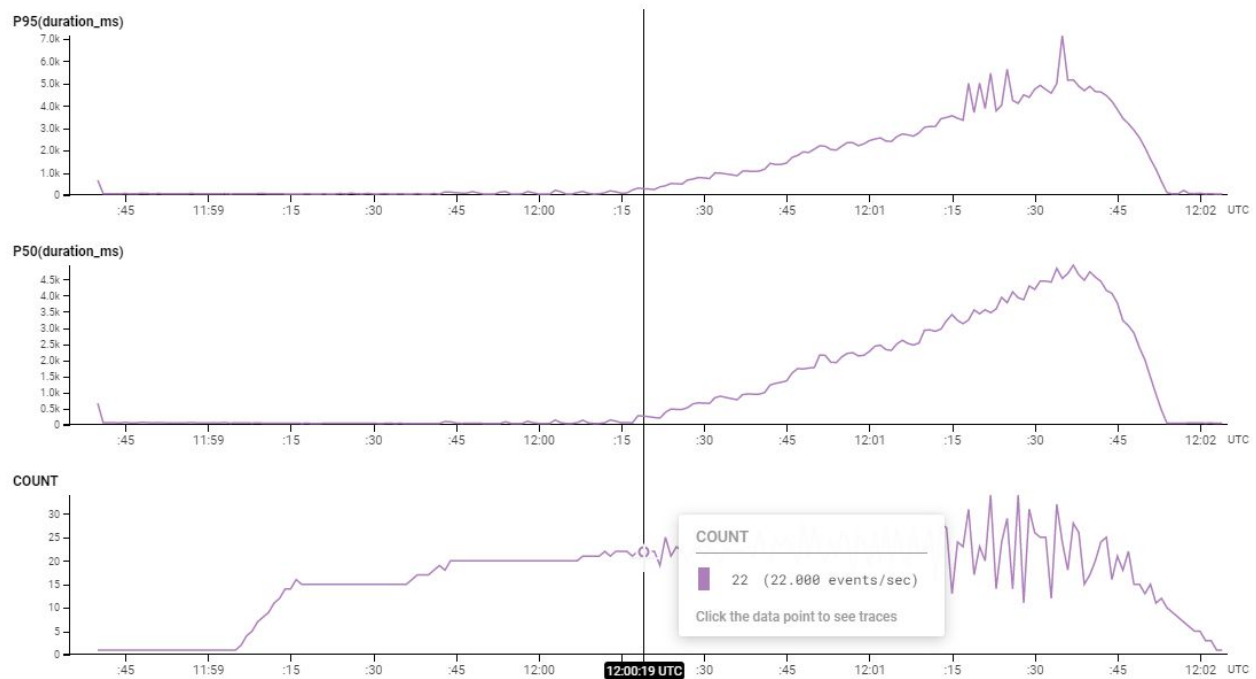


Figure 14: Honeycomb Result of Player Detail Request

Before the iter/sec increases to 22, the p50, p95 duration time is in a low value of less than 1s. Once the iter/sec larger than or equal to 22, the performance becomes unstable, the count is bouncing up and down and the P50, P95 duration is increasing.

Thus, we can conclude that by redesigning the searching logic of getting data for “player-detail”, **the “player-detail” page’s QPS before performance downgrade increases to 22, which is 70% better** than the previous improvement’s performance.

4.1.3.2 Redesign Database Searching Logic of Rendering “Match-detail”

The problem with the procedure of getting data for “match-detail” is slightly different.

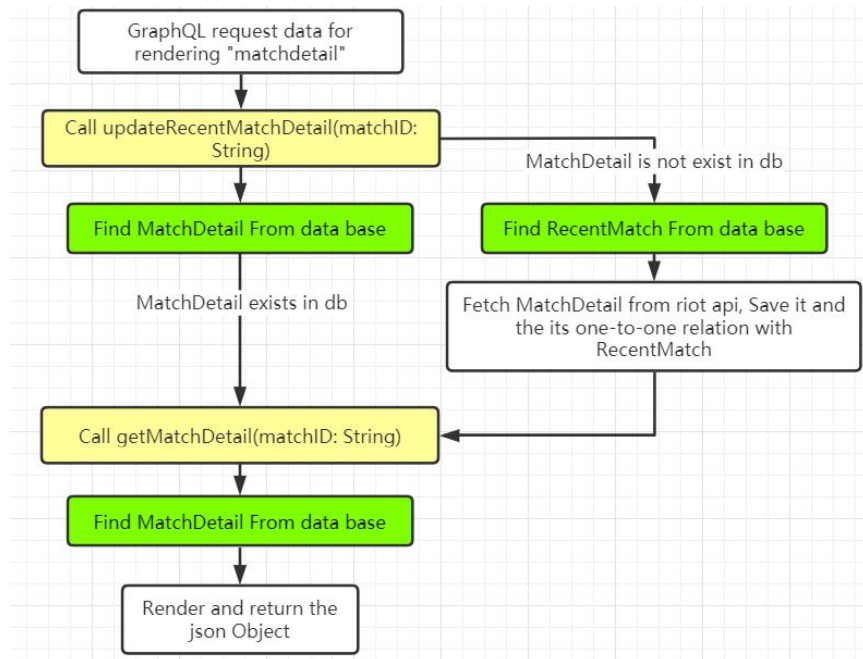


Figure 15: Initial searching procedure for rendering the “match-detail” page

The flowchart above is our initial searching procedure for rendering the “match-detail” page. When the required MatchDetail are present and valid(not outdated), it requires 2 SQL Finds, which is 4 SQL queries. If the MatchDetail data does not exist or needs updating, it requires 3 SQL Find, which is 6 SQL queries.

We can reduce one of the MatchDetail Find by reordering the function calls. The new flowchart of the searching procedure for rendering the “player-detail” page is shown below.

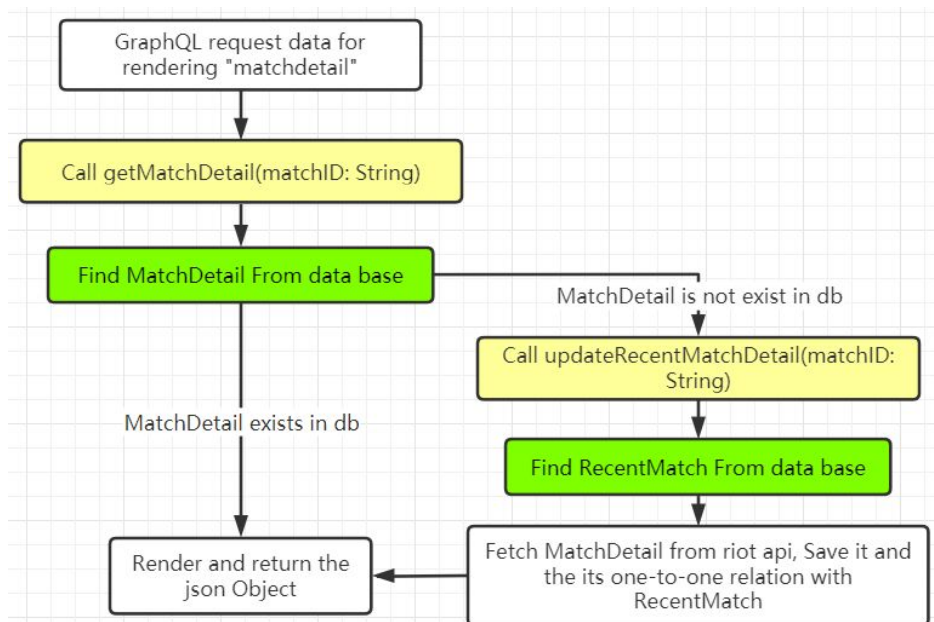


Figure 16: New searching procedure for rendering the “match-detail” page

In the new searching logic, updateRecentMatchDetail will only be called when the getMatchDetail could not find the MatchDetail from the database, Thus, inside the updateRecentMatchDetail, there is no need to check whether the MatchDetail exists in the database or not. Thus, the application only requires **2 SQL queries** (1 Find) for getting a pre-searched match detail.

To find out the new QPS before performance downgrade after the improvement, we change our “ramping-arrival-rate” testscript to ramp up in 10s and stay on its 20s with the target **1, 5, 7, 8, 9, and 10 iters/sec**. The Honeycomb result of the new load test is shown below:

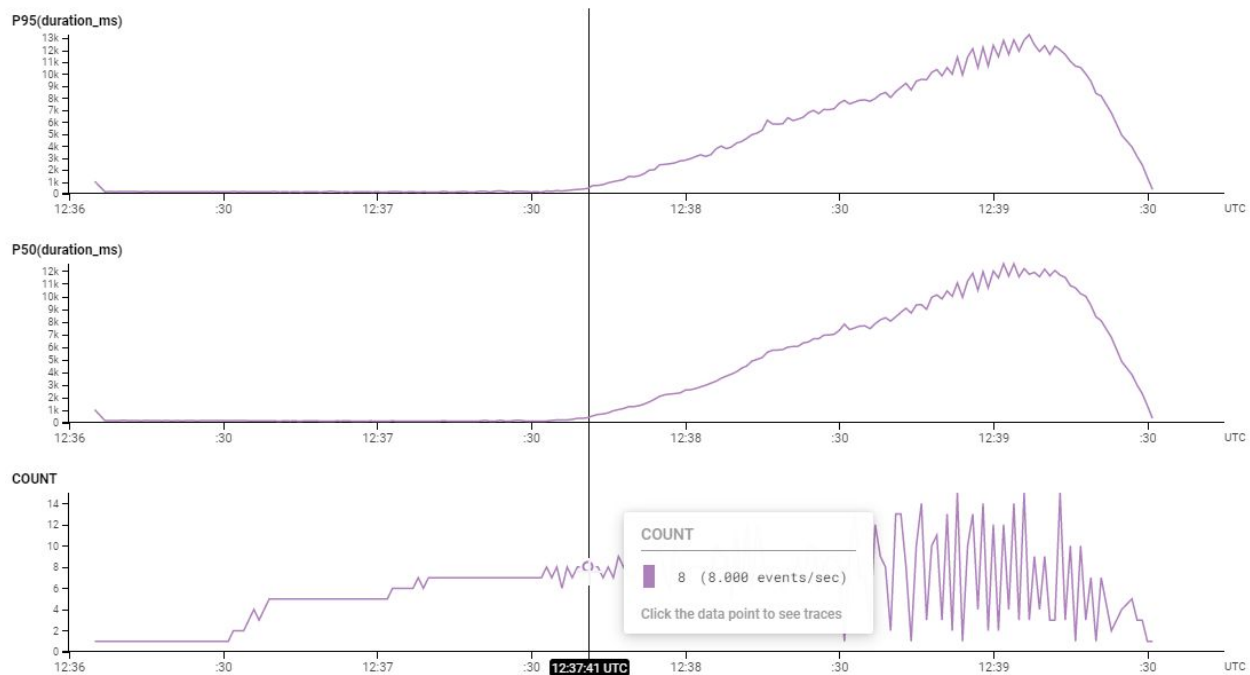


Figure 17: Honeycomb Result of Match Detail Request

Before the iter/sec increases to 8, the p50, p95 duration time is in a low value of less than 1s. Once the iter/sec larger than or equal to 8, the performance becomes unstable, the count is bouncing up and down and the P50, P95 duration is increasing. Thus, we can conclude that by redesigning the searching logic of getting data for “match-detail”, **the “match-detail” page’s QPS before performance downgrade increases to 7, which is 75% better** than the initial baseline performance.

4.2 Server-side Caching

Based on the result of the original load test where we only used MySQL as the database, we suffered numerous I/O timeouts leading to many request failures. We acknowledged that the bottleneck of server performance lies in disk I/O. To solve the problem, we decided to introduce Redis to cache frequently accessed data. Redis is an in-memory cache that stores data in memory instead of disk. Therefore, we expect that using Redis would bring us great performance improvement since memory I/O is faster than disk I/O.

We choose Redis instead of directly caching in the server memory so it can also be used as central storage after we scale the server horizontally. When we introduce more than one server instance to our system, the problem of data consistency in distributed systems also follows. By using Redis, all server instances can load data from one Redis database instance so data cached would be consistent for all server instances. If we directly use the server instance's memory, the popular data cached would be different for each server instance, which can not reflect the true popularity of the data.

Another reason we use Redis is that it can ensure data durability. Redis persists the cached data to the disk using RDB and AOF strategy. If we use the server's memory directly, our popular data would be lost when the server is down.

4.2.1 Caching Mechanism Design

As mentioned above, we cache popular data in Redis. We define the popularity of data as that the data has been queried more than 30 times within 30 minutes.

We maintain two hashmap data structures in the Redis for popularity checks. Another hashmap is maintained to cache the information associated with the player name or game ID (such as win rate, game participants, etc). These are data that we would return to the frontend.

Key	Value
Player Name	Number of times the key has been requested
Game ID	Number of times the key has been requested

Key	Value
Player Name	Last requested time (UNIX time)

Game ID	Last requested time (UNIX time)
---------	---------------------------------

When a request is received, the server first checks whether the requested player name or game ID is in the second map. If the second map contains the requested player name or game ID as a key and the time difference between the last updated time of the key and the current time is less than 30 minutes, the server then checks the first map. If the corresponding of the requested player name or game ID in the first map is greater than 30, the data satisfies the popularity. The data would be cached into Redis.

We also introduce a cache deletion mechanism when data is no longer “popular”. If the second map contains the requested player name or game ID as a key and the time difference between the last updated time of the key and the current time is greater than 30 minutes, both cached data and the data entry in the first hashmap would be deleted.

The data’s last updated time in the second map would be always updated to the current time.

A detailed caching store mechanism is illustrated below.

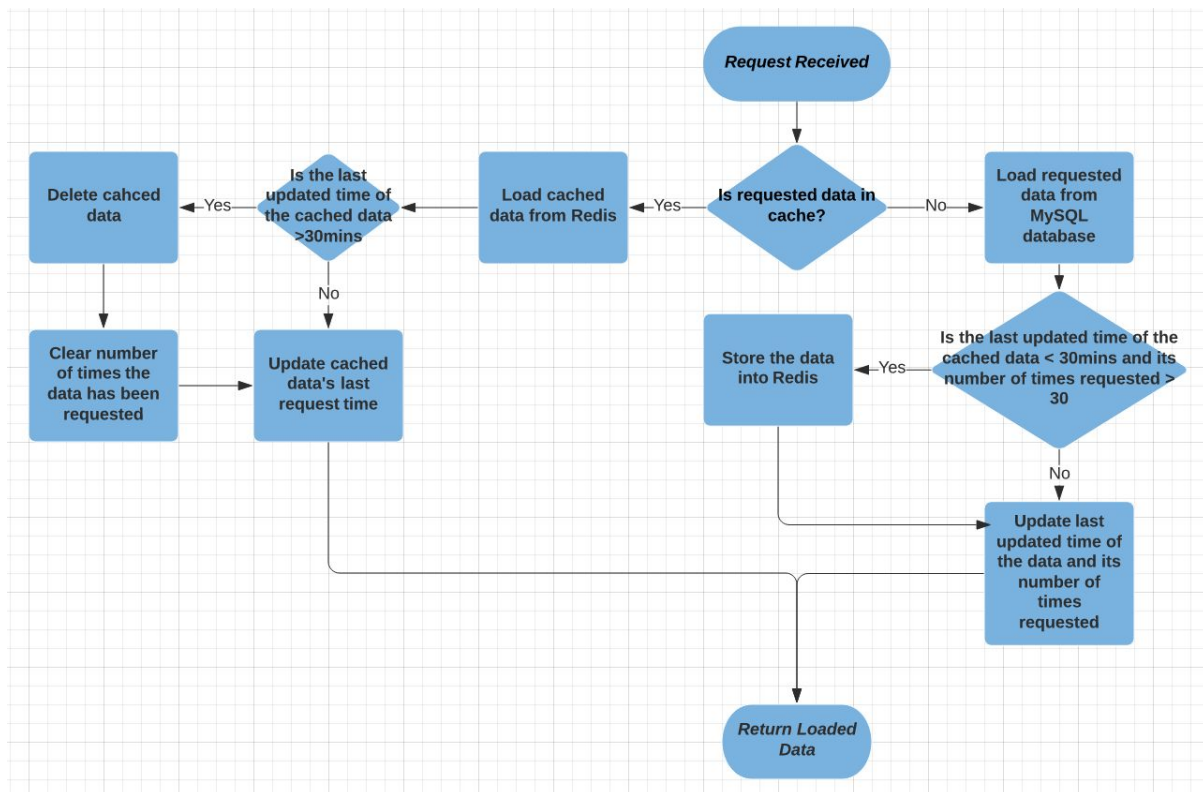


Figure 18: Logic Flow of Caching Mechanism

4.2.2 Performance (After Caching)

The following two figures show the honeycomb and k6 results of querying a certain player detail. We observe that the server begins to be downgraded since both P90(duration_ms) and P50(duration_ms) begin to increase significantly. **At this point, the number of events/second reached 42.**

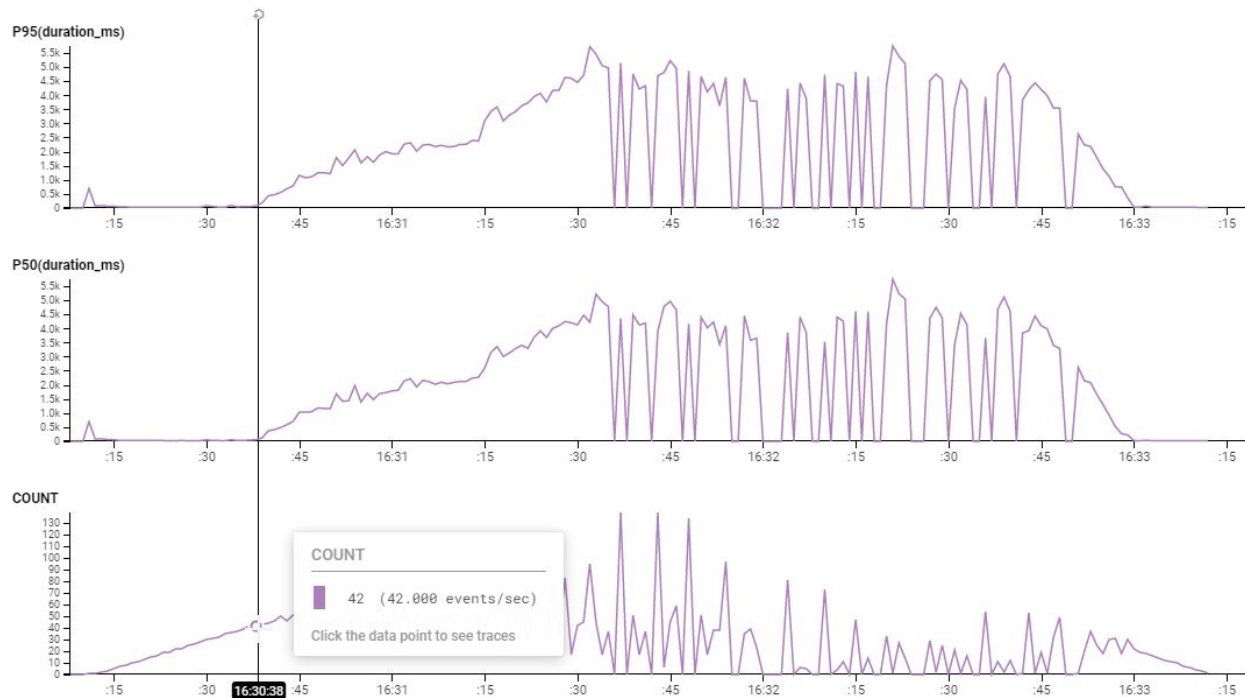


Figure 19: Honeycomb Result of Player Detail Request

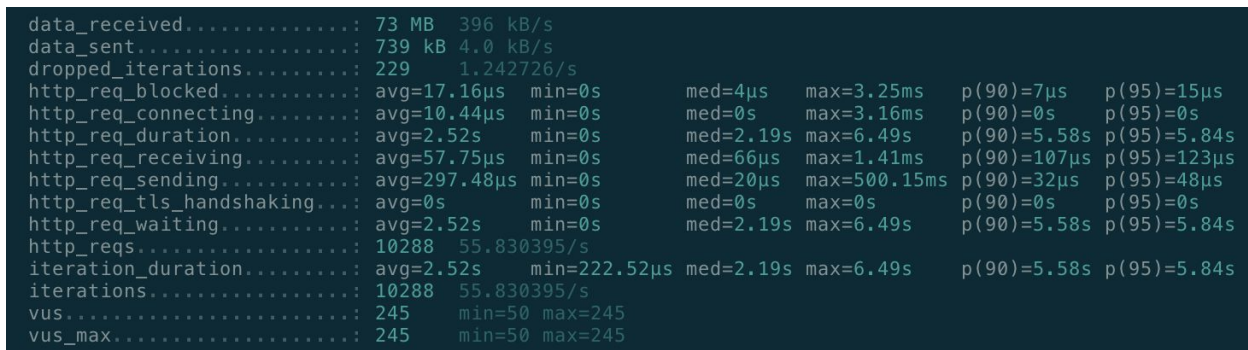


Figure 20: K6 Result of Player Detail Request

The following two figures show the honeycomb and k6 results of query a certain match detail. We observe that the server begins to downgrade since both P90(duration_ms) and P50(duration_ms) begin to increase significantly. **At this point, the number of events/second reached 18.**

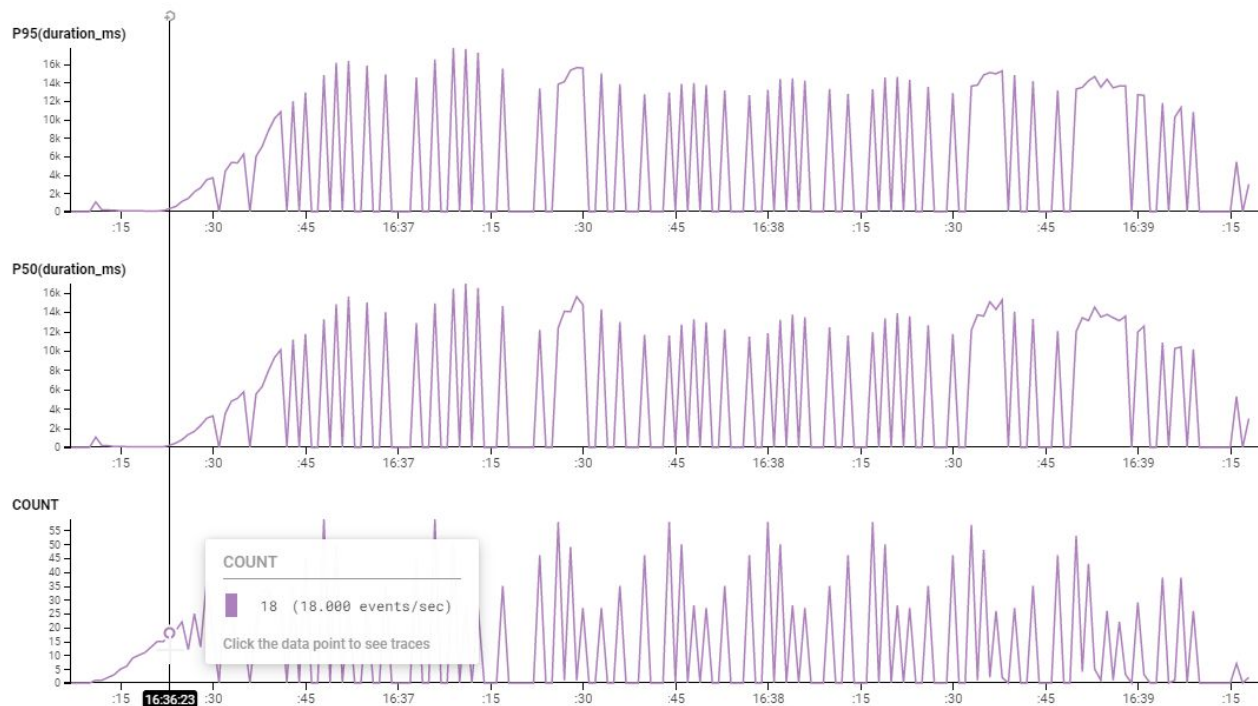


Figure 21: Honeycomb Result of Match Detail Request

```

data_received.....: 118 MB 608 kB/s
data_sent.....: 278 kB 1.4 kB/s
dropped_iterations.....: 220 1.136267/s
http_req_blocked.....: avg=10.21µs min=0s med=0s max=2.06ms p(90)=5µs p(95)=7µs
http_req_connecting.....: avg=6.85µs min=0s med=0s max=1.96ms p(90)=0s p(95)=0s
http_req_duration.....: avg=3.7s min=0s med=0s max=22.72s p(90)=16.65s p(95)=17.46s
http_req_receiving.....: avg=29.97µs min=0s med=0s max=1.81ms p(90)=115µs p(95)=139µs
http_req_sending.....: avg=1.33ms min=0s med=0s max=2.03s p(90)=26µs p(95)=33µs
http_req_tls_handshaking...: avg=0s min=0s med=0s max=0s p(90)=0s p(95)=0s
http_req_waiting.....: avg=3.7s min=0s med=0s max=22.72s p(90)=16.65s p(95)=17.46s
http_reqs.....: 10297 53.182441/s
iteration_duration.....: avg=3.7s min=196.69µs med=487.31µs max=22.72s p(90)=16.65s p(95)=17.46s
iterations.....: 10297 53.182441/s
vus.....: 253 min=50 max=253
vus_max.....: 253 min=50 max=253

```

Figure 22: K6 Result of Match Detail Request

Based on the above two load tests, we can conclude that the player detail route performs better than the match detail route in terms of the maximum number of events the server can respond per second before an obvious downgrade. On both load test figures, we see a large number of pulsations where events per second reached 50 - 110 happened after the breakpoint. We speculate that the server can't handle more requests and the number of requests it can handle has also become unstable. These pulsation points may be the main cause of server latency.

In terms of route latency, the player detail route performs better than the match detail route. In the normal phase, the request duration of the player detail route is around 30 milliseconds to

100 milliseconds while the request duration of the match detail route is around 100 milliseconds to 200 milliseconds. The player detail route also has a lower latency in the server downgraded phase. After the breakpoint, the request duration of the player detail route is around 5 seconds to 7 seconds while the request duration of the match detail route is around 10 - 16 seconds. The possible reason is that the match detail route requests more server resources (more images and JSON objects) than the player detail route.

4.2.3 User Behaviour Simulation Load Test

To get a more accurate result of the server downgrade breakpoint of each route, we only test a single HTTP endpoint in the last two load tests. To better measure server performance under real user behavior, we now use multiple HTTP endpoints to simulate users' behavior.

```
export default function () {  
  http.get('http://localhost:3000/app/player-detail/Yunbee2')  
  http.get('http://localhost:3000/app/player-detail/yassuo')  
  http.get('http://localhost:3000/app/player-detail/Revenge')  
  http.get('http://localhost:3000/app/match-detail/3693611915')  
}
```

The following figure shows the honeycomb results of query multiple player details and match details. We observe that the server begins to be downgraded since both P90(duration_ms) and P50(duration_ms) begin to increase significantly. **At this point, the total number of events/second reached 29.** Before the breakpoint, the total P50(duration_ms) of each route is about 281 milliseconds. After the breakpoint, the server starts to suffer a huge increasing number of requests. The response time continues to increase and the number of events in response becomes unstable, which is similar to the pulsations we observed in the above two load tests.

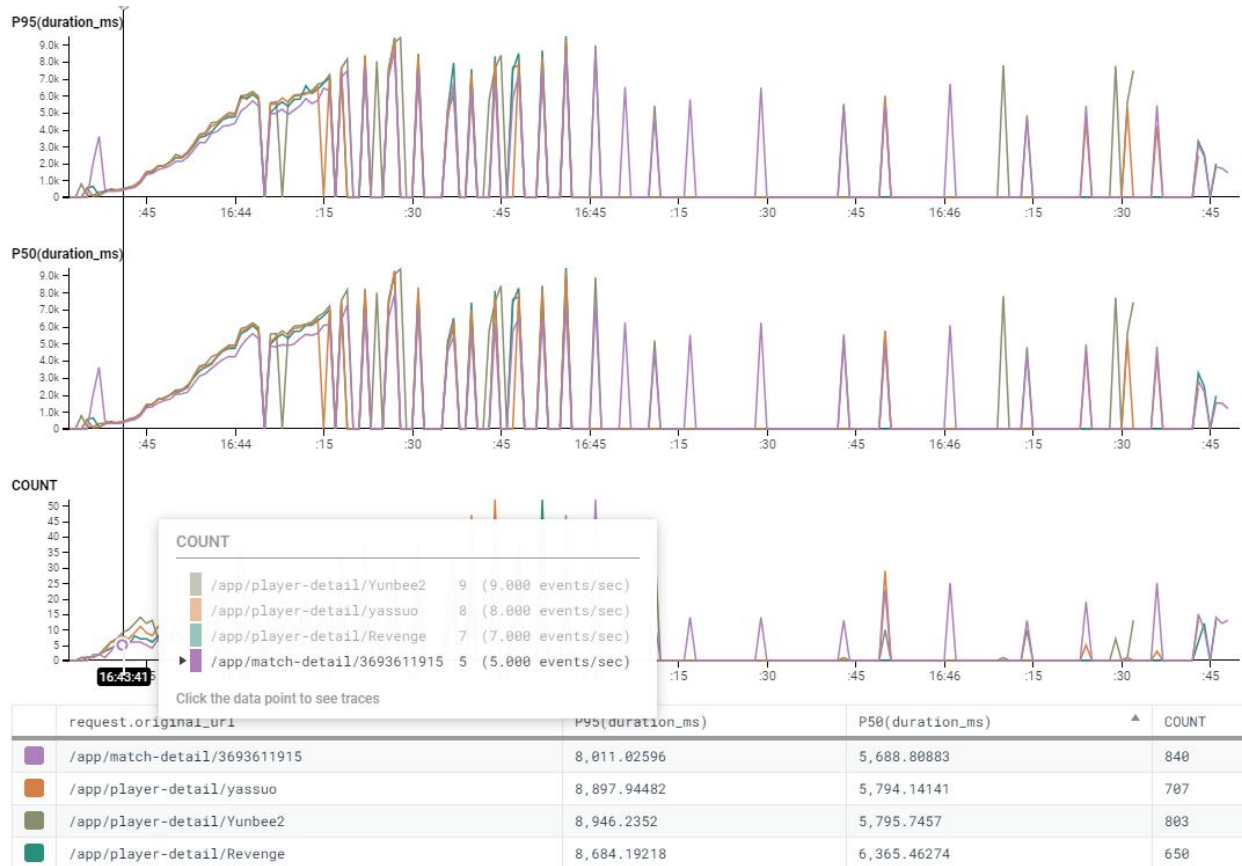


Figure 23:

5. Vertical & Horizontal Scalings

5.1 Scaling Tool & Settings

```
export default function () {
  http.get('https://twitchprime.cloudcity.computer/app/player-detail/Yunbee2')
  sleep(3 * Math.random() * 3)
  http.get('https://twitchprime.cloudcity.computer/app/player-detail/Revenge')
  sleep(3 * Math.random() * 3)
  http.get('https://twitchprime.cloudcity.computer/app/player-detail/Yassuo')
  http.get('https://twitchprime.cloudcity.computer/app/match-detail/3693611915')
  sleep(3 * Math.random() * 3)
}
```

The scaling tool being used in this part is also k6. Since the player names and match ids need to be valid, preset values were used. For each iteration, similar to the previous parts, the script visits two other players' player detail pages before visiting a final player detail page and a match detail page.

```

constant_vus_scenario: {
  executor: 'constant-vus',
  vus: 1000,
  duration: '120s',
},

ramping_vus_scenario: {
  executor: 'ramping-vus',
  startVUS: 0,
  stages: [
    { duration: '6s', target: 1 },
    { duration: '120s', target: 2000 },
    { duration: '60s', target: 0 },
  ],
  gracefulRampDown: '30s',
},

ramping_arrival_rate_scenario: {
  // name of the executor to use
  executor: 'ramping-arrival-rate',
  // common scenario configuration
  startRate: '2',
  timeUnit: '10s',
  // executor-specific configuration
  preAllocatedVUs: 1,
  maxVUs: 5000,
  stages: [
    //start only 1 instance at first to avoid multiple instances
    //to fetch the summoner data from riot api and save it to the the db
    { target: 1, duration: '10s' },
    { target: 5000, duration: '80s' },
    { target: 0, duration: '60s' },
  ]
}

```

Figure 24: the three scenarios

The testing scenarios being used are similar to the ones used in previous parts, and three scenarios were being used. The first scenario is a constant-vus with a target of 1000 VUs and a duration of 2 minutes. The second scenario is a ramping-vus with 3 stages. The first stage and the last stage are buffer stages, and the second stage has a target of 2000 VUs and a duration of 2 minutes. The third stage is a ramping-arrival-rate with 3 stages similar to the second scenario. The second stage this time has a target of 5000 VUs and a duration of 80 seconds.

5.2 Problems We Ran Into

After successfully deploying our application on AWS, we were able to access vertical and horizontal scaling options of the latest version of our application. Since the results are numerous, not all of the data is presented in the following sections.

We did two batches of load tests on different occasions due to uncertainties in the results. The first time load tests were being conducted, the Internet connection kept cutting out during the session. We accidentally forgot to provide the API key to the application, and the results

seemed unusual as well. Thus, another load test session was conducted the next day and the Internet connection was intact. Surprisingly, the patterns remained largely the same. Therefore, the results from both sessions are included.

5.3 Vertical Scaling

For some reason, we were not able to use memory sizes such as 512 MB and 1024 MB. Therefore, we used 2048 MB, 3072MB, and 4096 MB as the memory sizes along with a fixed 1024 CPU computing units. The results which contain mean and median latencies are as follows:

first session (in seconds)	2048 MB	3072 MB	4096 MB
constant-vus	56.18, 61	57.33, 61	58.44, 60
ramping-vus	38.65, 38.02	31.89, 23.33	33.79, 25.8
ramping-arrival-rate	39.06, 35.73	37.23, 33.12	32.88, 27.92
second session			
constant-vus	54.73, 58.93	64, 53.19	65, 71
ramping-vus	40.88, 42.62	35.75, 36.55	37.49, 38.58
ramping-arrival-rate	33.89, 29.71	35.91, 30.83	33.46, 29.39

Overall Vertical Scaling Performance

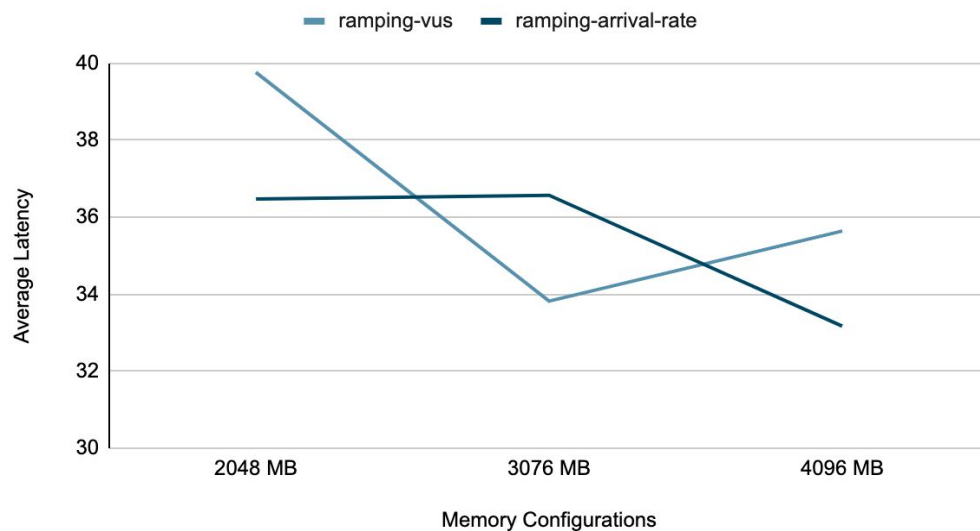


Figure 25: overall results of vertical scaling

For the first scenario which is constant-vus, both sessions show that vertical scaling isn't effective in this case, however, we noticed that repeatedly running the same load test leads to worse and worse results which may be due to the constant huge load on the server. Therefore, even though the vertical scaling seems to bring worse constant-vus results, the differences are minimal and we can say they are similar.

For the other two scenarios, overall we can see some significant improvements. Ramping-vus improves at least 10 percent when we increase the memory from 2 GB to 3 GB, but sees no improvement from 3 GB to 4 GB. Ramping-arrival-rate, on the other hand, has a constantly improving performance as the memory size increases but the performance gain isn't as noticeable as the second scenario.

Overall, vertical scaling improves the app's performance. For lighter loads, the three memory configurations performed similarly. Since we tested the deployed application with a heavy load, increasing the memory size allows the app to temporarily save up more requests without discarding them and thus handle heavy loads better. For our application specifically, 3 GB is more desirable since the performance gain from 2 GB to 3 GB is more notable compared to from 3 GB to 4 GB.

5.4 Horizontal Scaling

Horizontal scaling went through the same two iterations as vertical scaling as the load tests were back to back. Both iterations used 4 GB of memory and 1024 CPU units. The first iteration used 2 and 4 as the server counts compared to 1 while the second iteration used 3 servers since the patterns were obvious and we wanted to make sure. Since the three scenarios all share the same trend, the results in seconds for the third scenario are as follows:

first iteration	mean latency(s)	median latency(s)	iterations per second
1 instance	32.88	27.92	35.69
2 instances	76	74	13.66
4 instances	98	98	10.86
second iteration			
1 instance	33.46	29.39	48.7
3 instances	59.7	59.33	21.19

Overall Horizontal Scaling Performance Part I

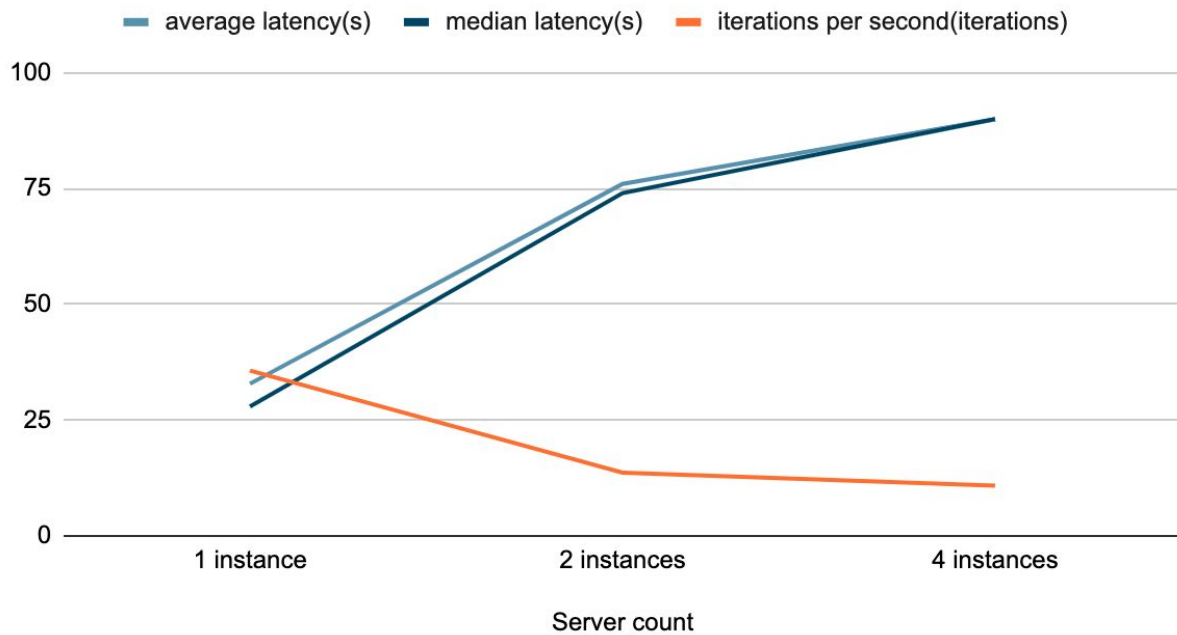


Figure 27: Results of the first iteration of horizontal scaling load test

Overall Horizontal Scaling Performance Part II

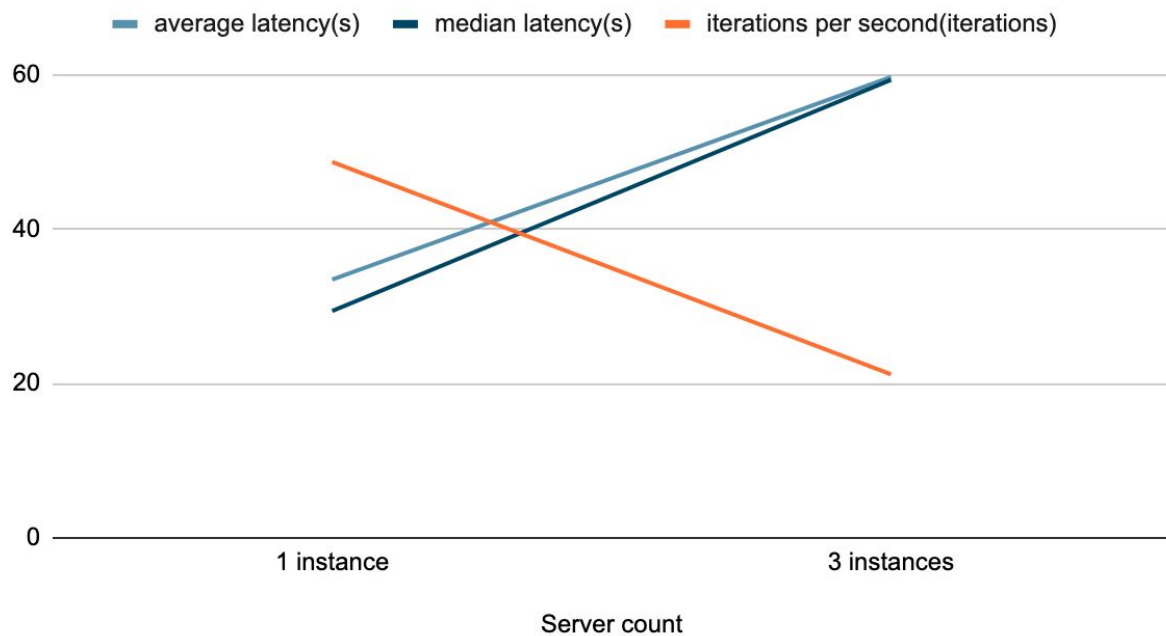


Figure 28: Results of the second iteration of the horizontal scaling load test

Both graphs show that horizontal scaling has the opposite effect regarding performance improvements. When the server instance goes up from 1 to 2, performance degrades drastically with latencies more than doubled. The second graph shows a similar story although to a lesser degree.

From what we observed during the testing process, for tests with more than one instance, very few iterations were completed during the majority of the allotted time frame and it ramped up towards the end. Our current implementation of server-side caching only comes in after a set number of iterations. Therefore, we think that our Typeorm database was having a huge problem handling requests from different server instances. The requests that were queued up because of the bottleneck had to time out which led to significantly worse performance.

A possible solution to this problem can be to redesign our cache to a first come first serve cache. It would cache the data of every request and the load tests don't really need to access the database any more after the first iteration. The basic database implementation we currently have still needed improvement as our application is quite data intensive.

6. Summary & Conclusions

6.1 Further Development

6.1.1 Caching the "Search-Name to Index" Map

In our final implementation, whenever the application searches for a Summer or a MatchDetail from the database, there are two SQL queries, the first one is looking for the index of the row the second one is getting the row with that index from the database

The first SQL query can be eliminated by utilizing server-side caching. If we can cache the map which key is the seachName (maybe the summonerName or the MatchId), and the value is the table index, then we can read the index of the database row from the cache and it is much more efficient than initiating a SQL query.

6.2 Conclusions

Although the application we built isn't the most scalable and needs further developments on scalability, we have learned a lot about testing scalability and ways to improve it. Throughout this quarter, we spent more time on building the application than scaling it and the consequences are evident. Horizontal scaling revealed problems that we had never thought about before, and we really don't know exactly how to solve them given the very limited time we have left. Nonetheless, we are very thankful for the fun course material offered to us. Sorry for writing such a long paper.

References

[1] Riot Developer Portal. developer.riotgames.com.