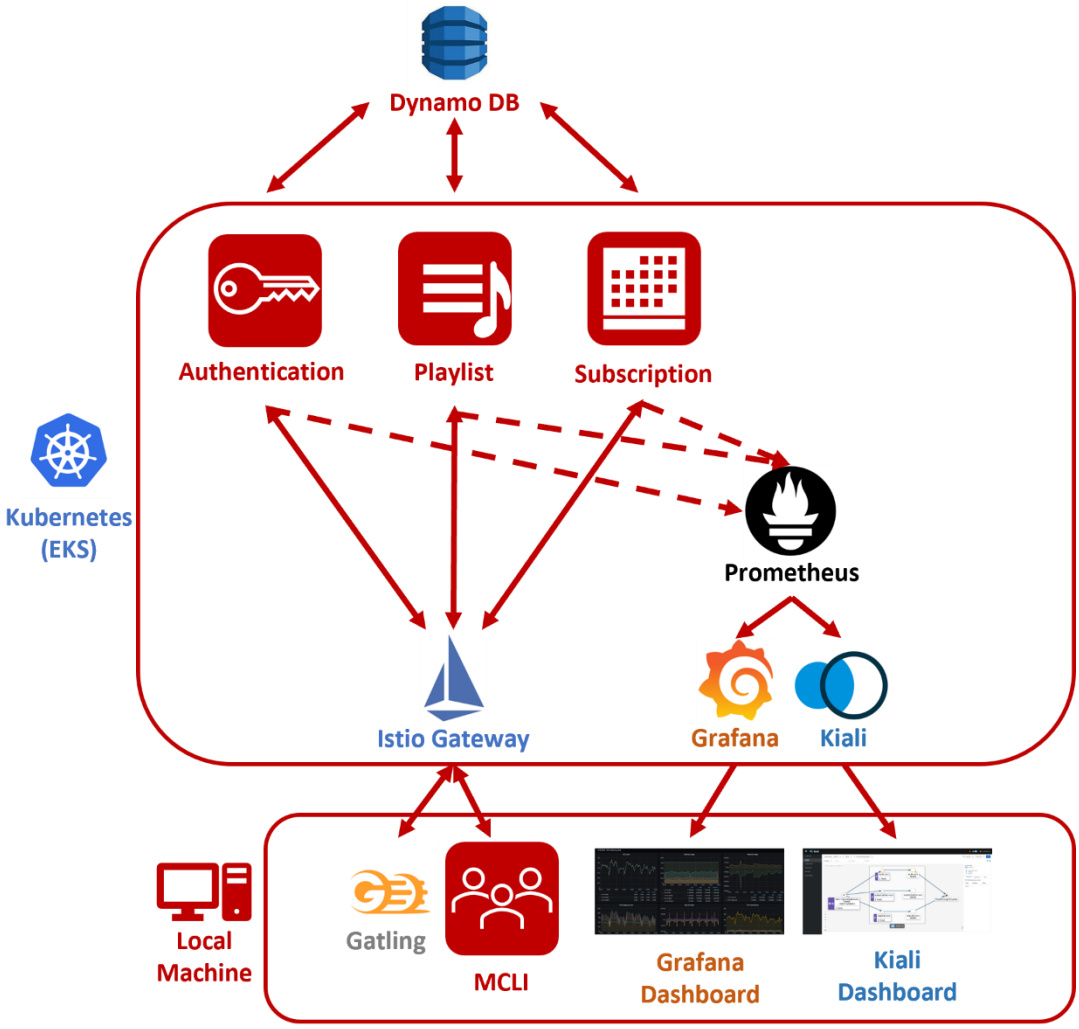
CMPT 756 Kube Squad/Square Term Project Report

Team Name	Kube Squad/Square				
CourSys team URL	https://coursys.sfu.ca/2022sp-cmpt-756-g1/groups/g-kube-square				
GitHub project repo URL	https://github.com/scp756-221/term-project-kube-squad				
Team Member Name	Aidan Vickars	Rishabh Kaushal	Anant Awasthy	Karthik Srinatha	Abhishek Nair
Email (SFU)	avickars@sfu.ca	rka73@sfu.ca	asa404@sfu.ca	ksa166@sfu.ca	asn4@sfu.ca
Github ID	avickars	risha-bhkaushal07	asa-404	Karthiksrinatha	abs990
Additional Notes					

Summary of Application

Due to the large number of members in our group, as well as general interest we developed three original micro-services that work together to form a playlist application with authentication and subscription services. The application is visualized in the image below.



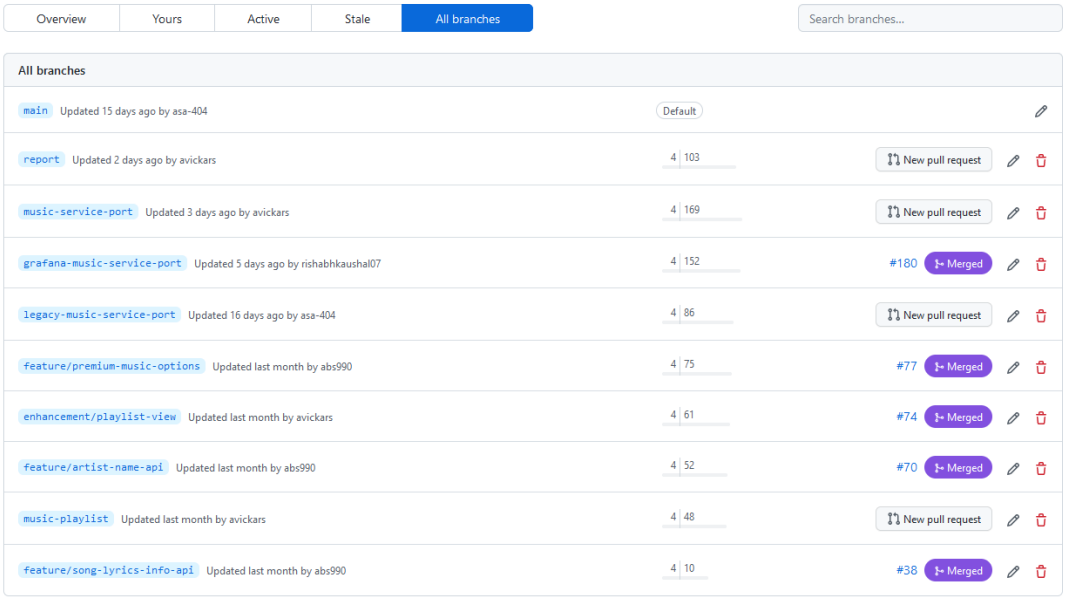
Playlist Application

As stated above, the application was designed to run as three containerized micro-services that run on Kubernetes. Beginning with the Authentication service, this service allows users to create or login into their account respectively and subsequently access the Playlist service that will be described shortly. Like any normal application, accessing features often requires a paid subscription. Our application is no different; accessing the features in the Playlist service requires a paid subscription. We have implemented this in the Subscription service, where after logging in or registering for an account using the Authentication service users can choose their preferred subscription option and subscribe by adding a credit card that passes through a simulated validation. Once users have logged in and subscribed, they can access the Playlist service that contains a variety of features. These features include creating, viewing, and editing playlists, as well as finding information related to specific songs such as the lyrics, genre, and artist. All three services use Dynamo DB for storage and independently query the database to access information like user accounts, song information, subscription statuses etc.

From the client side, to make requests to each service we used two tools. The first is a modified version of the given MCLI application that has been fitted to make the appropriate requests to each service. The second is Gatling where we have created test suites to place varying amounts of load on each service. To assess how well the application responds to load, we have configured Prometheus, Grafana and Kiali to gather and visualize metrics accordingly. This will be discussed further in our analysis below. Finally, to distribute the load for each service we have created a gateway into the cluster using Istio that acts as the load balancer for the entire system.

GitHub Guide

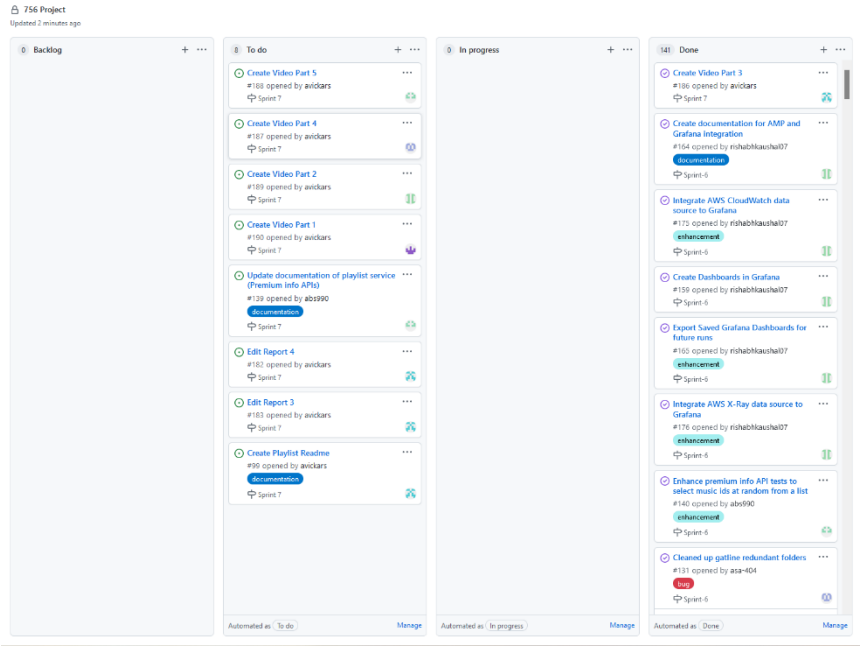
Beginning with the structure of the GitHub repository, the development branch is "music-service-port" that contains the most recent working build of the application. It is expected that any team member that merges their work into this branch, verifies that the branch remains in an error free condition. It should be noted that the production branch is the "main" branch. Furthermore, we utilized branching extensively during the development process. When a team member was actively working on a feature or bug, they would create a new branch to use as their working copy and merge their work into the "music-service-port" branch when ready. The remaining branches are visualized in the image below.



Screen Shot of Remaining Branches in GitHub

To augment the development process, we made extensive use of GitHub's projects and logged every item of work as an issue in the "756 Project" that contained our Kanban board. In this board, there are four separate columns that contained the backlog, to do, in progress and done items. In our typical workflow, team members would freely add issues to the backlog that would likely need to be completed. In our weekly scrum backlog items are moved into the to do column when as a team we feel they need to be completed in the coming sprint and are subsequently assigned to a team member. At the same time, other backlog items were often removed from the board entirely when the team decided they were obsolete.

Whenever a team member embarked on a new piece of work, they moved the corresponding issue into the “in progress” column or if the issue had not already been created, they created the issue first before continuing. Finally, once the issue was completed, it would be moved to the “done” column. At the time of writing there are 155 issues in the “done” column. It should be noted, we do attempt to link our commits to the relevant issues, however we have found that remembering to do this was difficult. The Kanban board is visualized in the image below.



Screenshot of Project Kanban Board

Observations

Reflection on Development

What did you observe from applying and using the scrum methodology? What worked well? What didn’t? What surprised you?

After applying and using scrum methodology, we made three significant observations. The first is that it allowed us to more accurately assess our currently velocity towards completing the project. Due to the nature of course work that often comes in waves, team members development capacity often fluctuated significantly week to week. By having weekly scrums, we were able to properly define the definition of done for the upcoming sprint such that it was achievable. Furthermore, in the case a team members capacity changed on short notice (ex: assignment was pushed back), having short sprints allowed us to pivot quickly.

The second observation we made was that the consistent cadence that scrum often requires did not work. As mentioned above, team members development capacity would fluctuate significantly week to week. As a result, there were sprints where the entire team’s development capacity was effectively zero and the resulting sprint goal was nothing. Thus, the team was unable to maintain a consistent cadence that at times defeated the purpose of scrum because there was simply no point in having one since no one had capacity to perform any work.

The third and final observation we made was that it normalized the behaviour of demonstrating one's work to the team. By applying scrum, it normalized the expectation of demonstrating the completed items from the last sprint. We found this to be particularly useful because it served as both an opportunity for everyone to demonstrate their work, but also as an opportunity to teach others how to utilize their completed work. An example of this was the containerization of the application where each service was ported to work as containers. After demonstrating the applications running as containers, each team member became familiar with the process of building and running the application.

Reflect on the readings over the course of the term. What ideas were you able to apply? How did these turn out?

Beginning with the second reading that featured a comparison between mono and poly repositories, we applied a mono-repository with great success. Due to the size of the application, using a poly-repository would have created additional overhead in managing the location of each service. In contrast, by using a mono-repository we were able to easily create a single source of truth that contained the application and facilitated rapid deployment and iteration. This rapid deployment was further enhanced by the "infrastructure as code" ideas that were also presented in the second reading. To implement this, we wrote a comprehensive make-file that contains every command required to deploy the application both locally and in the cloud. This enabled a consistent deployment practice that eliminated all confusion around the deployment of the application. It should be noted that our use of containers and make-files to facilitate the deployment of the application captured ideas from the third reading as well that discussed containerization and make-files.

In the fourth reading, the deployment of applications using Kubernetes is discussed. We found that deploying our application to Kubernetes was a relatively straight forward process. The low-level implementation details that Kubernetes manages such as the scheduler, creating individual components and others meant that deploying our application was as simple as executing a few commands. We do note, that configuring Istio as discussed in the tenth reading was a more difficult process. This was due to the poor documentation that Istio provides. To be succinct, the documentation gives configuration examples, but does not explain what the individual components in YAML files are.

Moving on to the seventh reading that discussed micro-services, we found implementing our application as three micro-services enabled rapid parallel development. Because of the isolated nature of each service, we were able to rapidly complete all of them in the span of approximately two weeks with minimal integration issues and in a continuous fashion as described in reading eight. By having each developer merge their work into the "music-service-port" branch, we were able to continually develop the application while maintaining a working and stable version.

If you have professional experience with scrum, how did your team perform in comparison to past teams?

Our team performed both well and poorly compared to our professional experiences with scrum. From a cadence perspective our team performed quite poorly. As discussed previously we were unable to maintain a consistent cadence that contrasts with our professional experiences. In our previous professional environments, our focus would often be on a single project and as a result we were able maintain a consistent cadence towards completing items in each sprint. On a positive note,

during times when our team was actively developing, utilizing scrum enabled us to ensure there was no duplication of work. This was in stark contrast to some of our professional experiences where we would often perform a significant amount of work only to find later that there was existing work that could have easily been utilized instead. An example, of this is writing large portions of code only to find out that there was existing code base that could have been easily ported over to the current project.

Reflection on Operating the Application

As is shown below in the analysis, our application was able to function well in a steady state with load of approximately 1000 simultaneous users when a single service was targeted with the following cluster configuration:

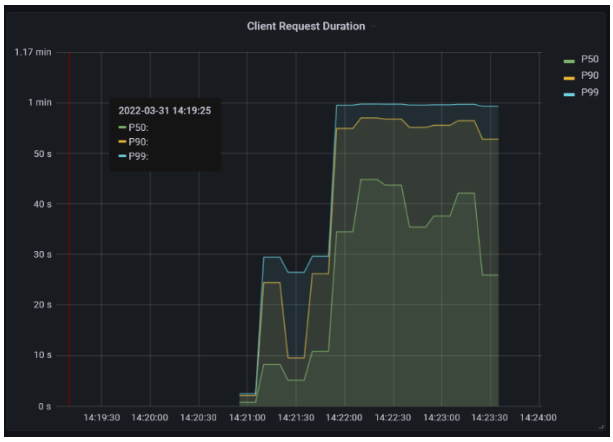
Configuration:

- Target Service: Authentication
- Replication Factor: (8 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Number of Simultaneous Users: 1000

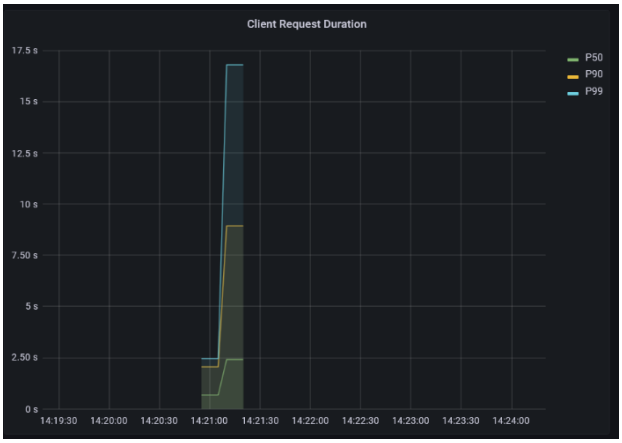
However, when all three services were targeted, the load had to be decreased significantly to approximately 50 simultaneous users to eliminate request failures. When all three services were targeted, the following cluster configuration was used:

Configuration:

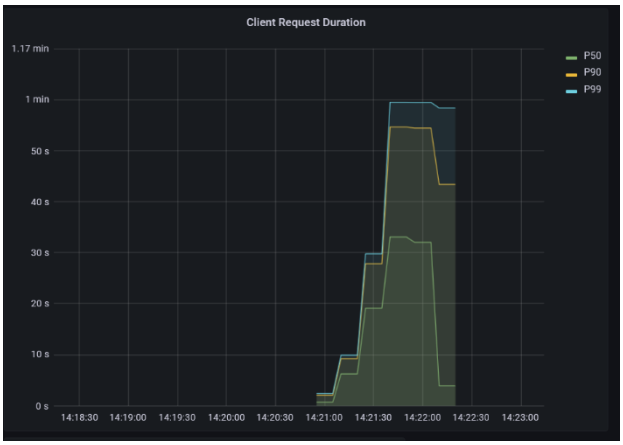
- Target Service: Authentication
- Replication Factor: (3 - Authentication, 3 - Subscription, 3 - Playlist)
- Number of Nodes: 3
- Number of Simultaneous Users: 50



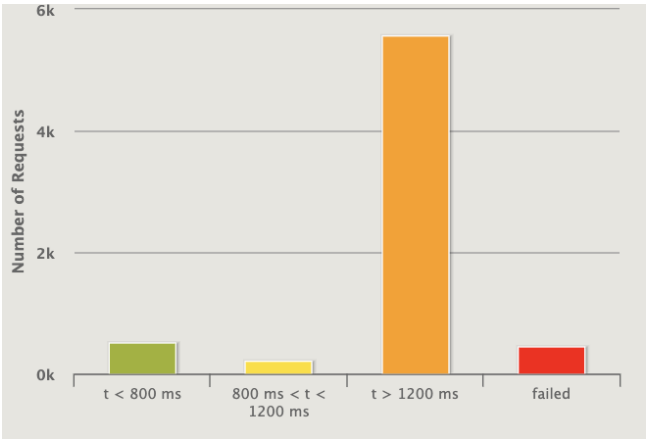
Playlist Latency



Subscription Latency



Authentication Latency



Request Outcomes

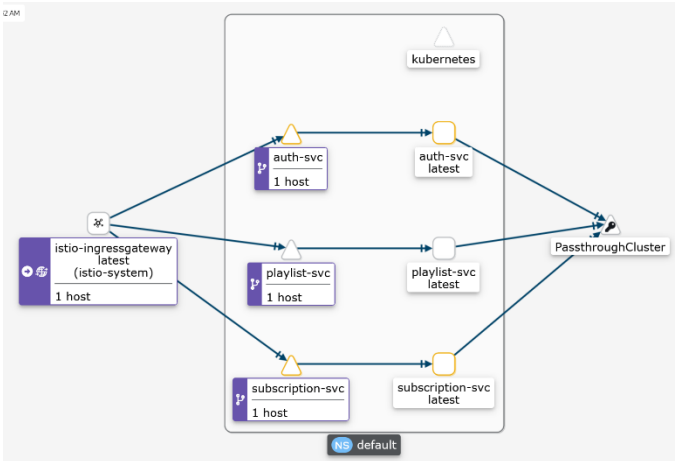
Using this configuration, we found from the above visualizations that most requests to the application succeeded, but the latency remained high in comparison to when a single service was targeted as shown in the scaling analysis below. We suspected this was due to an insufficient cluster configuration. This was confirmed in the scaling analysis below when only a single service was targeted at a time and the targeted service was also scaled up. In these cases, all failures were eliminated, and latency was reduced. Furthermore, it should be noted that the failed requests shown in the request outcomes visualization are time out errors that are a direct result of the high latency.

Analysis

To assess how our application scaled while under varying load from Gatling, we leveraged Prometheus, Grafana and Kiali. Prometheus was used to gather various metrics, while Grafana and Kiali were used to produce visualisations of the services. To be succinct, we used Grafana to gain a greater understanding of the latency in the services, and Kiali to understand how requests were passing through the system. We also used the reports from Gatling after each load test to assess the failure rates of the requests.

Request Flow Analysis

Beginning with the visualization produced by Kiali that is shown below, we can see how traffic flows through our application. Initially, all traffic flows into the gateway we defined using Istio that then distributes the load to each service accordingly. The reader will note that for each service there appears to be two pods or services. This is just a visualization of the routing configuration that was defined for each service. The multiple pods or services for each service can be interpreted as one. Once a request has reached a service, the service then makes corresponding requests to another service outside the cluster. These are simply requests made to Dynamo DB. However, since it exists outside the cluster it is not managed.



Application Request Flow

Scaling Analysis

Now that we understood how requests flowed through the application, we assessed the scaling of the application under varying loads. To do this we defined two tests that were applied to each service. These tests are defined below.

Test A:

Number of Users: 1000

Flow:

For every endpoint in the targeted service:

- *Make requests*
- *Pause for 3 seconds*

Test B:

Number of Users: 10 000

Flow:

For every endpoint in the targeted service:

- *Make requests*
- *Pause for 3 seconds*

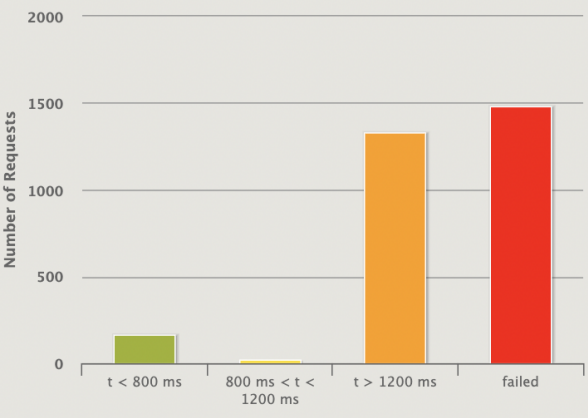
To assess how the application scaled, we applied tests A and B multiple times and recorded the results. The results and their corresponding explanation of every test are presented in the sub sections below. The learnings from previous tests are carried over to subsequent tests. Note, to eliminate Dynamo DB from our analysis, we configured Dynamo DB using AWS's "no capacity planning and pay-per-request pricing" module. As a result, Dynamo DB automatically scales itself.

Test 1

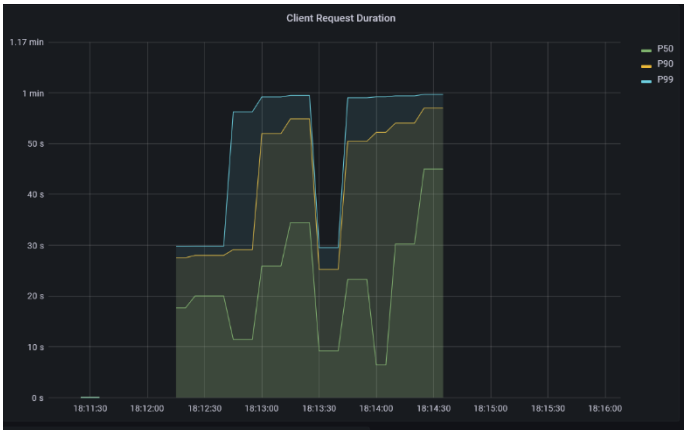
Configuration:

- Target Service: Authentication
- Replication Factor: (1 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 2
- Test: A

Results:



Authentication Request Outcomes



Authentication Latency

Learnings:

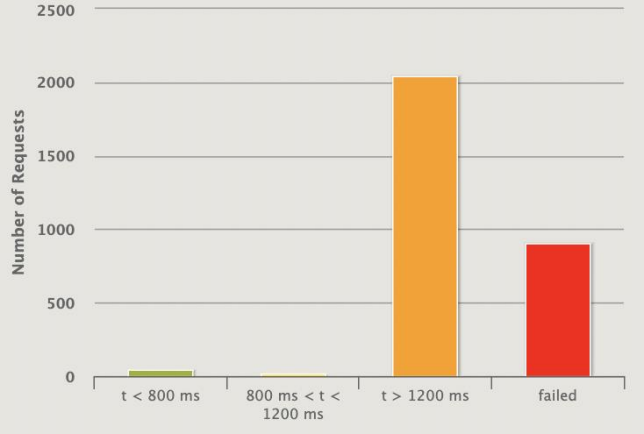
From the report produced by Gatling we can see that there was a very large number of failed requests. We found that most of these failures were due to timeout errors. This corresponds with the latency vitalization where even the 50th percentile was almost 50 seconds. These results clearly indicated that the cluster configuration was insufficient. To remedy this, we tuned the application by varying the replication factor of the Authentication service from 2 - 8. The results of the test with a replication of 8 applied to the Authentication service is shown in test 2.

Test 2

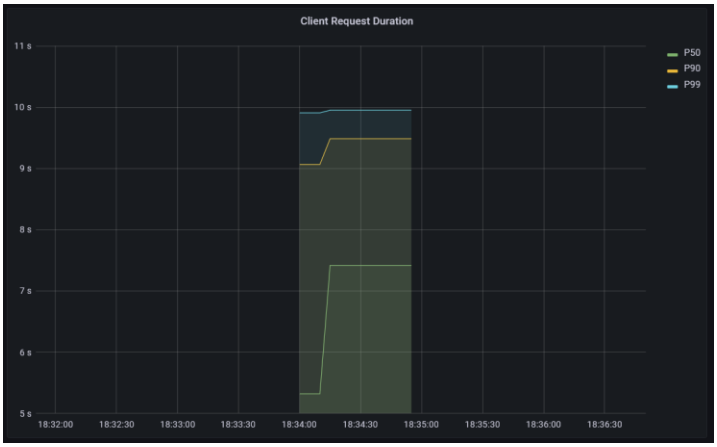
Configuration:

- Target Service: Authentication
- Replication Factor: (8 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 2
- Test: A

Results:



Authentication Request Outcomes



Authentication Latency

Learnings:

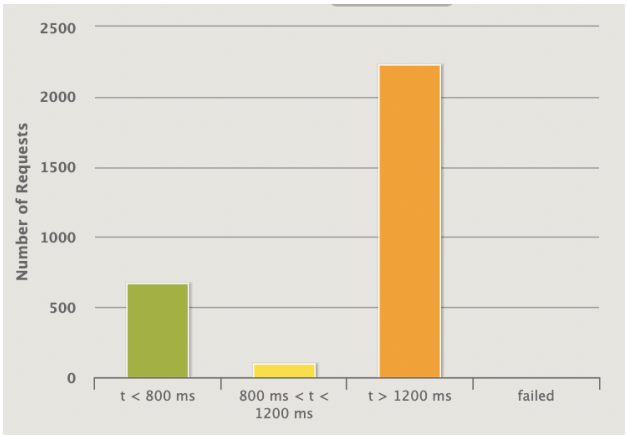
We can see that after increasing replication factor to 8 for the Authentication service, we saw a significant decrease in the number of request failures. This is reflected in the visualization of the latency where the 50th percentile is approximately 7.5 seconds. However, we wanted to eliminate all failures. Since after tuning the replication factor, the Authentication service was still experiencing failures, we took this as an indication that the nodes in the cluster may be saturated. To remedy this, we increased the number of nodes in the cluster by one and re-applied the test in test 3.

Test 3

Configuration:

- Target Service: Authentication
- Replication Factor: (8 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: A

Results:



Authentication Request Outcomes

Authentication Latency

Learnings:

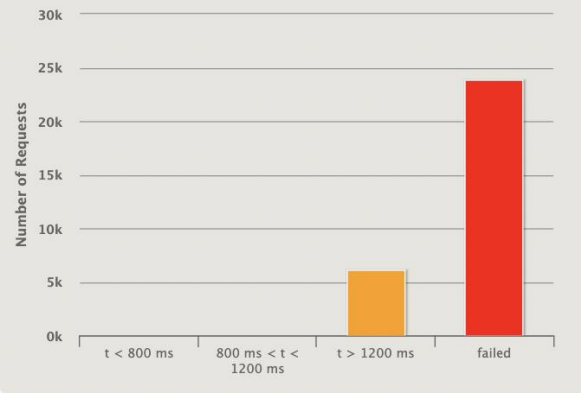
We found that once we increased the number of nodes, we were able to eliminate all failures. Furthermore, we can see a significantly improved latency in both images. In the Gatling report, there are significantly more responses with a latency under 800 milli-seconds. This is reflected in the Grafana visualization where even the 99th percentile is under 10 seconds aside from the single large spike. Now, that the application had no failures and lower latency, we applied test B below.

Test 4

Configuration:

- Target Service: Authentication
- Replication Factor: (8 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: B

Results:



Authentication Request Outcomes

Learnings:

In this test the number of concurrent users and requests per second increased by factor of 10. We can clearly see that the increased load resulted in a substantial increase in failures. The application did such a poor job in handling the load that Grafana would not actually produce the latency visualization and as a result is not shown. This clearly shows that the current configuration of the cluster was insufficient in both the replication factor and the number nodes. To remedy this, we did increase the number of nodes to 4 and increased the replication factor over varying sizes and re-executed the test. However, we saw no improvement. Due to the 10x increased load in test B relative to test A, we theorize that to achieve significantly better results we would need to increase the configuration of the cluster by a factor of ten as well both the replication factor and number of nodes. However, due to cost we did not attempt this. Though, we do note that from our results in tests 1, 2, and 3 we can deduce that improving the latency and reducing the number of failed requests is simply a matter of scaling the number of nodes and increasing the replication factor.

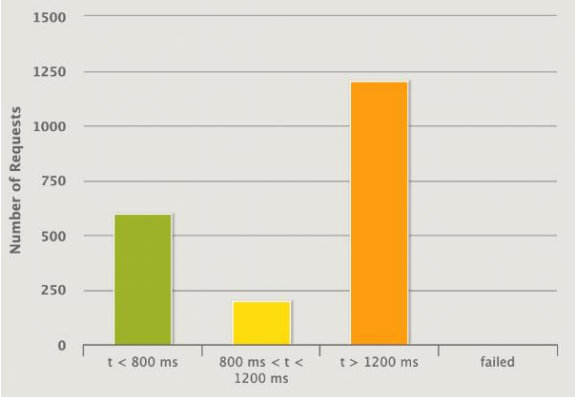
Test 5

Configuration:

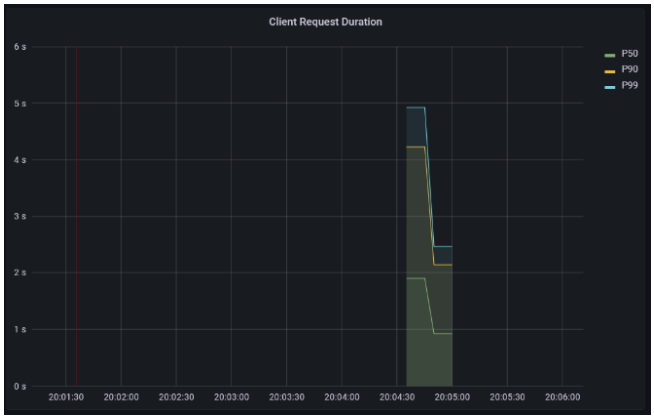
- Target Service: Subscription

- Replication Factor: (1 - Authentication, 8 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: A

Results:



Subscription Request Outcomes



Subscription Latency

Learnings:

Now that the configuration for the Authentication service has been tuned, we applied what we learned to the Subscription and Playlist services. As a result, we achieved extremely similar results to what was achieved in test 3 with the exception that the Subscription service was targeted instead of the Authentication service. We found that the service did not experience any request failures, but still had a latency that was well above 1 second. We were not able to improve this in any meaningful fashion with the current number of nodes in the cluster.

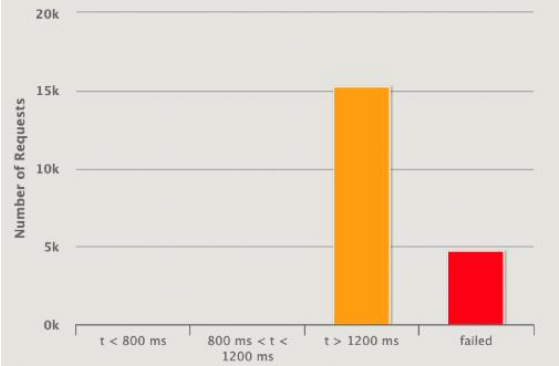
Furthermore, this configuration was applied on the Playlist service but in the interest of minimizing repetition we do not explicitly state the results. This is because the results were almost identical with the notable difference in that the latency was significantly higher. This was unsurprising because the requests made to the playlist service required additional filtering in Dynamo DB or outright required the entire list of songs contained in the database to be returned.

Test 6

Configuration:

- Target Service: Subscription
- Replication Factor: (1 - Authentication, 8 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: B

Results:



Subscription Request Outcomes

Learnings:

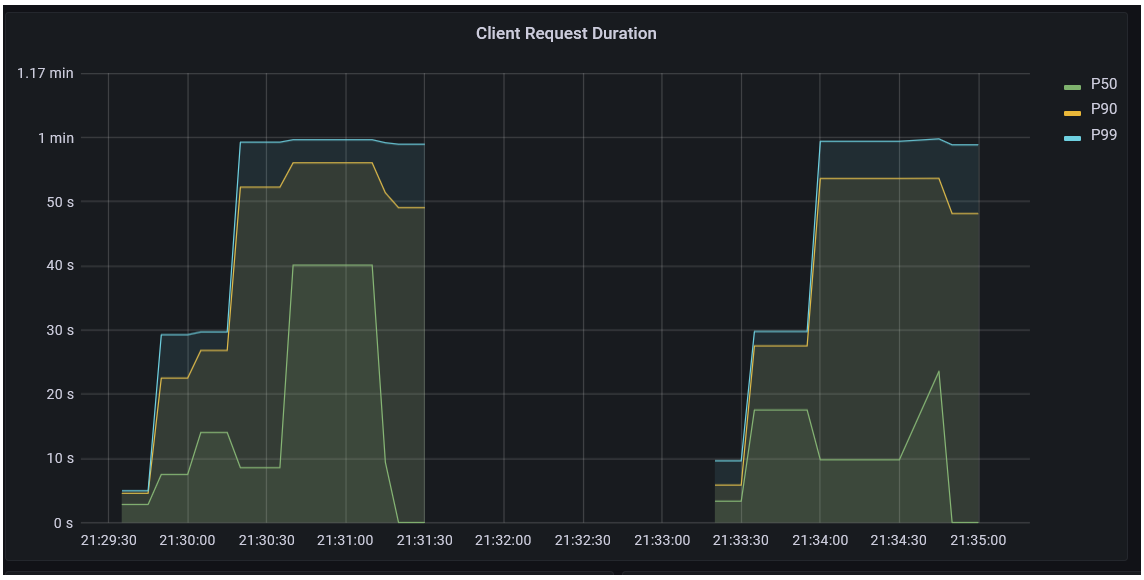
To further validate the results we found in test 4 where we applied the much heavier test B on the Authentication service, we applied test B on the Subscription service. Unsurprisingly, the results were almost identical. In short, the heavier load resulted in a substantial increase in request failures from the client side and Grafana was again unable to produce the latency visualization due to the increased load. This test was also applied to the Playlist service with similar results however to minimize repetition we do not visualize the results here. Thus, we conclude again that the current cluster configuration was insufficient. To adequately handle this level of load, the cluster would require additional scaling of the nodes, and an increased replication factor for each service.

Test 7

Configuration:

- Target Service: Authentication – *with autoscaling*
- Replication Factor: Initially (1 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: A

Results:



Authentication Latency

Learnings:

To incorporate what we have learned in the previous tests, we configured horizontal autoscaling in the cluster for each service according to the number of requests per second. Using a very low target of 1 request per second to encourage fast scaling, we executed test A on the Authentication service. We found that while autoscaling does work it takes a significant amount of time for Kubernetes to recognize that it should increase a service's replication factor. This can be seen in the latency visualization above where after executing test A on the authentication service two times, the latency did not improve. This was because Kubernetes required a lengthy and sustained load to recognize that it should scale the service. To be succinct, it took two executions of test A for Kubernetes to scale the Authentication service from 1 to 8 replicas. Once, the replication factor increased to 8, we achieved similar results to that from test 3. In short, we found that the autoscaling performs very well for clusters that experience varying but sustained levels of load. But it does not perform well on clusters that experience short and intense bursts of load. This is because Kubernetes requires a significant amount of time to recognize and scale accordingly.

Failure Analysis

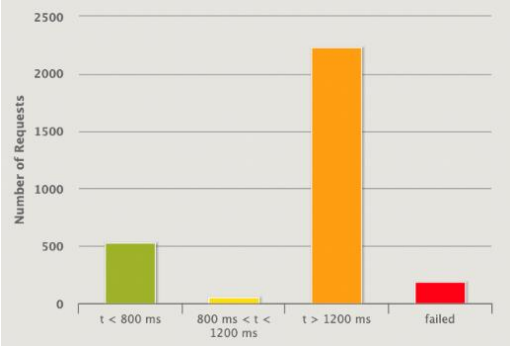
To assess what happens to the application when it experiences failures, we applied tests A multiple times while applying two types of failures.

Test 8

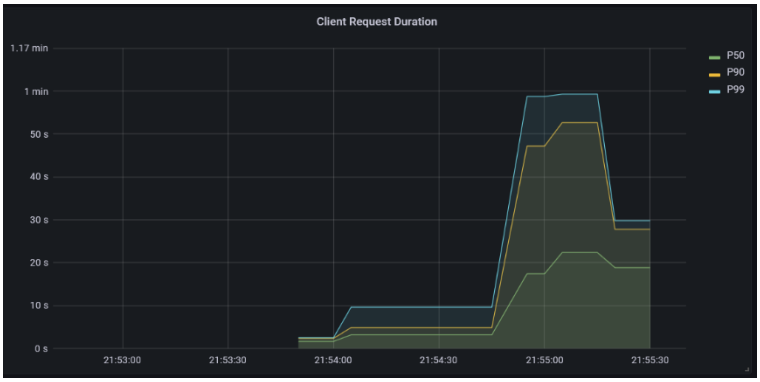
Configuration:

- Target Service: Authentication
- Replication Factor: (8 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: A
- Failure: Pod Failure

Results:



Authentication Request Outcomes



Authentication Latency

Learnings:

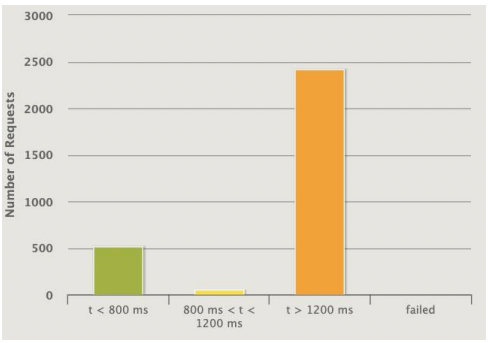
During this test, we randomly deleted two pods containing the Authentication service while the test was running. We found that while the service did not crash, it did cause request failures from the increased latency. Looking at the latency plot, we can see the exact moment the pods go down due to the significant spike in latency. Furthermore, we can also see towards the end of the test that the service begins to recover. This is due to Kubernetes self healing capability. Immediately after the pods were deleted, Kubernetes created two new pods to take their place. As a result, there was only a temporary increase in latency and request failures.

Test 9

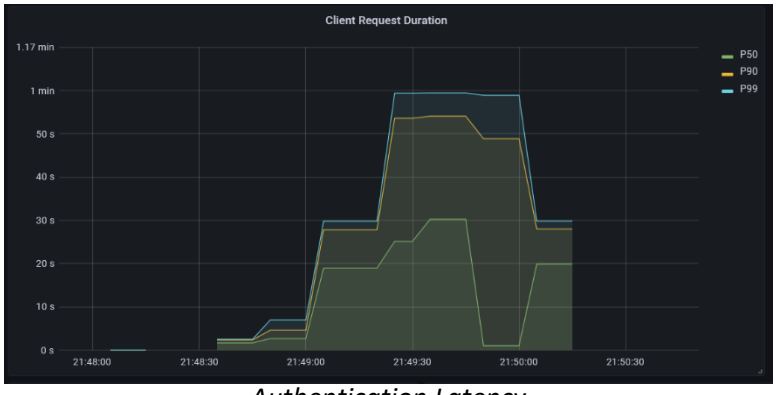
Configuration:

- Target Service: Authentication
- Replication Factor: (8 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: A
- Failure: Request Delay

Results:



Authentication Request Outcomes



Authentication Latency

Learnings:

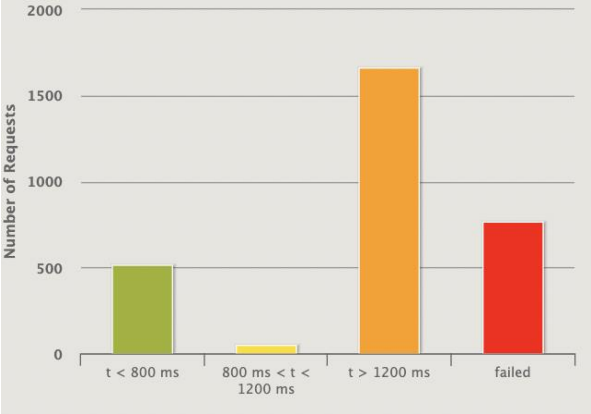
During this test, we configured 25 % of the requests made to the authentication service to have a delay of 25 seconds. We found to no surprise that this significantly increased the latency of the service relative to that seen in test 3. However, it did not result in any request failures. This indicated that the service using this configuration and under this load has some resistance to delayed responses.

Test 10

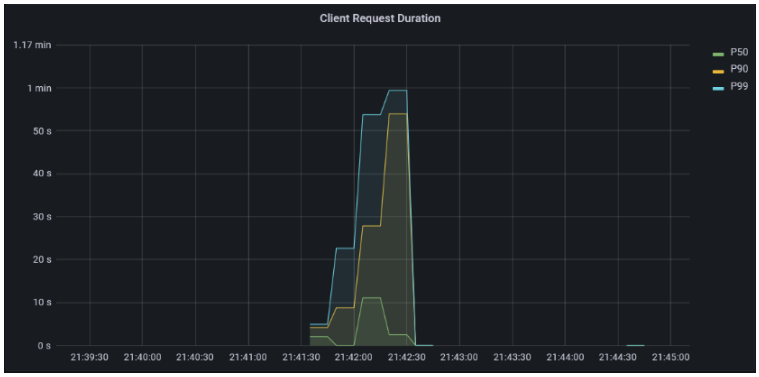
Configuration:

- Target Service: Authentication
- Replication Factor: (8 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: A
- Failure: Request Abort

Results:



Authentication Request Outcomes



Authentication Latency

Learnings:

During this test, we configured 25 % of the requests made to the authentication service to be aborted. Unsurprisingly, almost exactly 25 % of the requests failed. However, we that these failed requests resulted in no meaningful improvement in latency relative to the results in test 3.

Failure Remedies

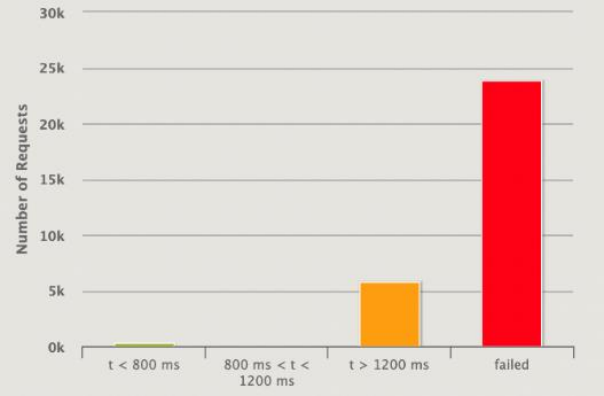
In this section, we explored remedies that can be applied to failures. In more detail, in test 10 we allowed for failed requests to be retried to reduce the number of failed requests. In tests 11 and 12 we configured a circuit breaker to intentionally cause all in progress requests to fail when the load on the application exceeded some threshold. Note, we did not try configuring request retries as done in test 11 with request aborts as done in test 10. This is because request retries with request aborts do not work together in Istio as discussed here.

Test 11

Configuration:

- Target Service: Authentication
- Replication Factor: (8 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: B
- Remedy: Request Retry

Results:



Authentication Request Outcomes

Learnings:

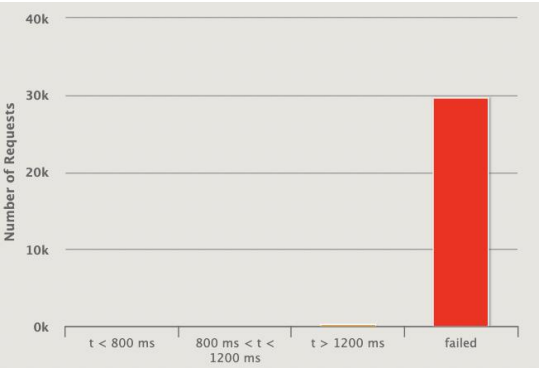
Recall when we applied test B on the Authentication service in test 4, this resulted in a substantial number of failed requests due to the significant load of test B. To remedy this, we configured failed requests to be retried 3 times with a 5 second time out in-between. However, we found that this resulted in no improvement. In fact, it slightly increased the number of failed requests by twenty. We suspect the reason for this is that because the application was already under such significant load, retrying the failed requests only increased the load such that they would just fail again.

Test 12

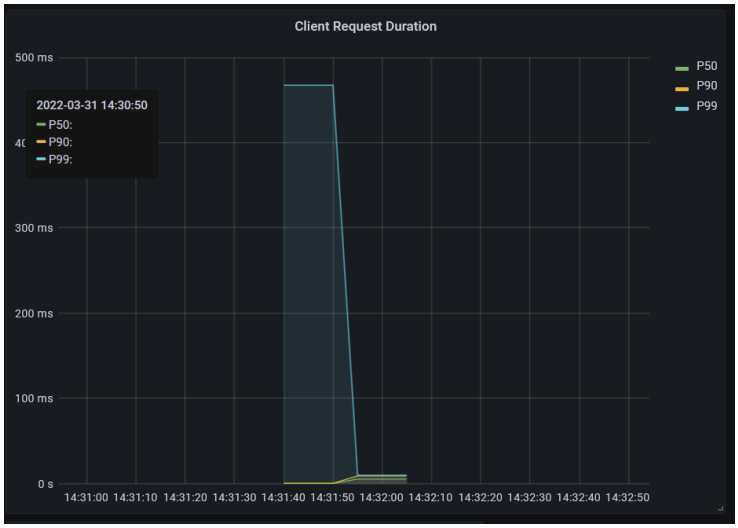
Configuration:

- Target Service: Authentication
- Replication Factor: (8 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: A
- Remedy: Circuit Breaker – Max Number of Pending HTTP Requests = 20

Results:



Authentication Request Outcomes



Authentication Latency

Learnings:

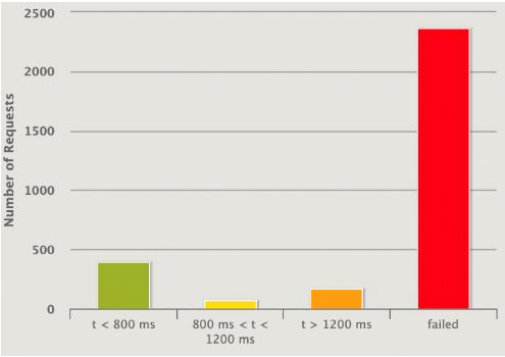
In this test we explored the use of circuit breakers to stop the application from being overloaded. Initially, we set the maximum number of pending HTTP requests to 20 to demonstrate the function of a circuit breaker. We can clearly see in the latency visualization when the circuit breaks because there is a steep drop off in latency when all the pending http requests are aborted. This is reflected in the gatling report where there is a small number of requests that succeeded that correspond with the section with high latency in the latency visualization.

Test 13

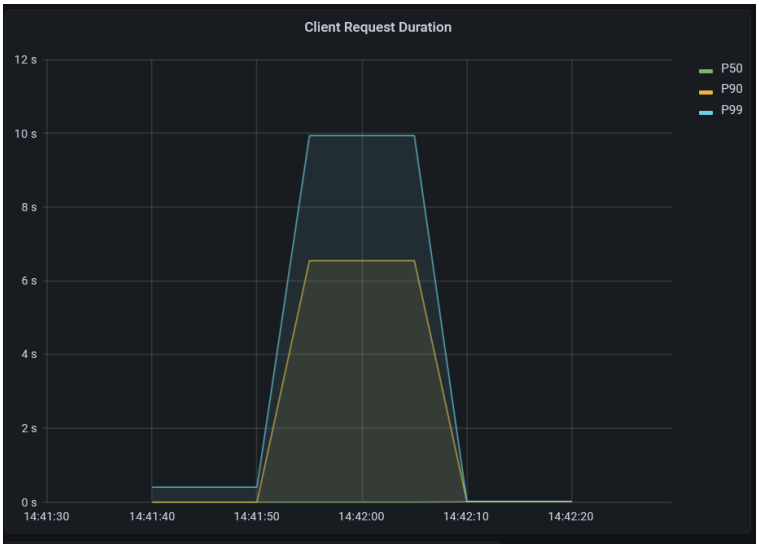
Configuration:

- Target Service: Authentication
- Replication Factor: (8 - Authentication, 1 - Subscription, 1 - Playlist)
- Number of Nodes: 3
- Test: A
- Remedy: Circuit Breaker – Max Number of Pending HTTP Requests = 500

Results:



Authentication Request Outcomes



Authentication Latency

Learnings:

To further explore the use of circuit breakers, we explored its ability to restrict or allow traffic into the application by loosening the maximum number of pending http requests the application would allow. After allowing additional pending requests, we unsurprisingly saw that a greater number of requests succeed before the circuit breaker was tripped which can be seen in the steep drop off in latency. However, because we allowed additional pending requests, we can see that the latency of the requests significantly increased compared to the latency in test 11. This clearly illustrates the potential usage of circuit breakers. Where developers can define a circuit breaker that will break when the backlog of requests to the service grows too large. This could be useful to allow the service time to scale up and subsequently improve latency without becoming bogged down from the backlog of requests. Furthermore, we can also clearly see the use of circuit breakers to create a more resilient application. Where developers can define the circuit breaker to trip when the application has reached its maximum scaling ability. This could again ensure that the application does not remain bogged down in a backlog of requests.