Cloud Lab

Cloud Computing - From Infrastructure to Applications
M2 MoSIG

Team Members:

Abbas Khreiss
Youssef Itani

Base steps

Deploying the original application in GKE

Briefly explain what is this Autopilot mode?
The Autopilot mode allows the cloud provider to manage our own cluster's configuration, scaling, security and other pre configured security. Utilizing the Autopilot mode, we are able to optimize our cluster since it provides compute resources based on the service's manifest files in order to figure out the best configuration.

Why does it hide the problem of not having enough resources to run all the required pods in our cluster?
The autopilot hides this because it does autoscaling when we add pods. It provides new nodes for those Pods, and automatically expands the resources in your existing nodes based on need.

Autopilot Reference:
Autopilot overview | Google Kubernetes Engine (GKE) | Google Cloud

To deploy the original application in GKE, we used the terraform configs that were already used in the provided repo. However, we changed the configs to use the GKE standard version instead of the autopilot. We did this change after encountering problems when implementing monitoring and canary releases due to security constraints enforced by the autopilot.

The terraform configuration provisions 4 e2-standard-2 for the cluster. It then uses Kustomize to construct the deployment manifest file and apply them. Afterward, it waits for all pods to become ready.

Deployment

In order to deploy, you must execute the following:

First, enable the required services

```
$ cd scripts
$ ./01-enable-gcp-required-services.sh
```

Next, set the project ID

```
$ ./02-set-gcp-project-id.sh {PROJECT_ID}
```

After setting the project, you proceed to deploy the cluster (~ 7 minutes)

```
$ ./03-deploy-boutique-cluster.sh
```

When the cluster is done deploying, you can extract the IP address of the frontend load balancer in order to access the site

```
$ ./04-boutique-get-frontend-ip-address.sh
```

Analyzing the provided configuration

Deployment Configuration:

- **apiVersion**: Specifies the Kubernetes API version for the Deployment resource.
- **kind**: Deployment: Declares that this YAML document is a Deployment.
- **metadata**: Contains metadata about the Deployment
- **spec**: Describes the desired state for the Deployment.
  - **selector**: Specifies how the Deployment identifies which Pods to manage. In this case, it will manage the app with the name "adservice."
  - **template**: Describes the Pod template that will be used to create new Pods.
    - **metadata**
    - **spec**:
      - **serviceAccountName**: Associates the Pod with a service account named "default."
      - **terminationGracePeriodSeconds**: Time Kubernetes waits for the Pod to terminate gracefully before terminating it forcefully
      - **securityContext**: Defines security-related settings for the Pod.
        - fsGroup, runAsGroup, runAsNonRoot, runAsUser: Define user and group permissions for the Pod.
      - **containers**: Defines the containers within the Pod.
        - **name**: Specifies the name of the container ("server").
        - **securityContext**: Security settings for the container.
        - **image**: Specifies the container image to use.
        - **ports**: Defines the container's ports to expose (exposes port 9555).
        - **env**: Environment variables of container (port with the value "9555").
        - **resources**: Resource requests and limits for CPU and memory.
        - **readinessProbe** and **livenessProbe**: Health checks for the container

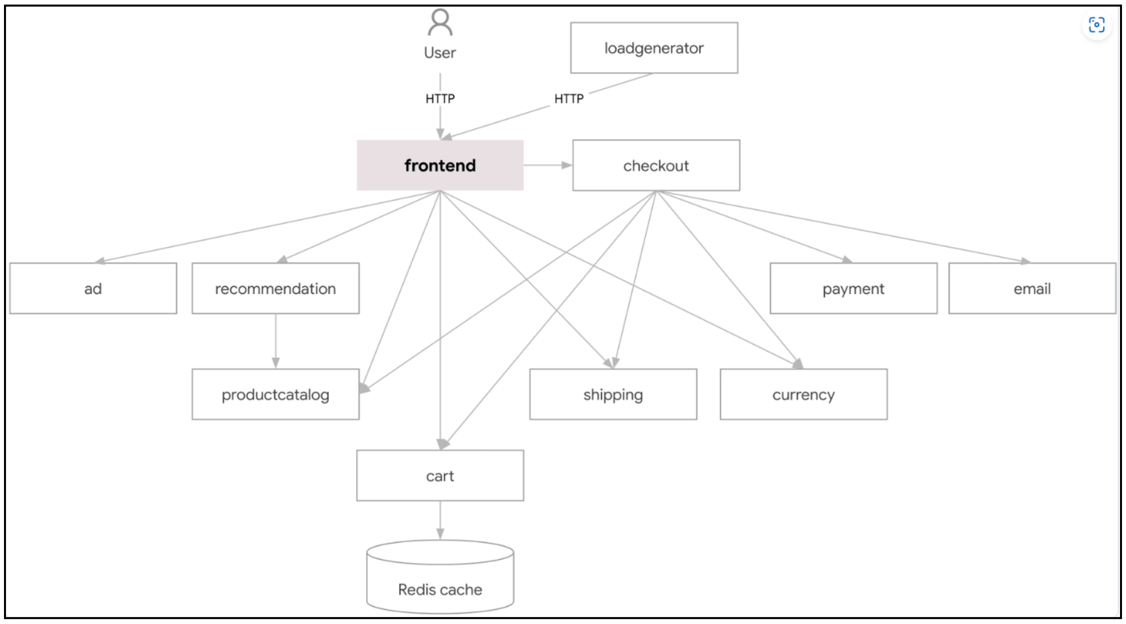
Service Configuration:

- **apiVersion**: Specifies the Kubernetes API version for the Service resource.
- **kind**: Service: Declares that this YAML document is a Service.
- **metadata**: Contains metadata about the Service
- **spec**: Describes the desired state for the Service.
    - **type**: ClusterIP: Sets the Service type to ClusterIP, making it accessible only within the cluster. Can also be used to make the service accessible to the internet
    - **selector**: Specifies how the Service discovers the Pods it should load balance traffic to.
    - **ports**: Ports exposed by the Service.
        - **name**: Name of the port ("grpc").
        - **Port**: Port on which the Service will be available.
        - **targetPort**: Specifies the port on the backend Pods to which the Service will route traffic.

Targeting a minimal deployment

We inspected the dependency between the services of the online boutique and we found that the load generator, Ad service, recommendation service and email service are not necessary for the deployment. Therefore, we removed them from the deployment. To verify that the site is still functional, we performed manual tests, and tests using Locust.

As we are deploying our application using Terraform and Kustomize, we created a Kustomize component called minimal-deployment to remove the unnecessary services.



Deploying the load generator on a local machine

In order to deploy the load generator on a local machine, we created a computer instance and installed the necessary packages required to obtain the Dockerfile from the repository, build the image and run it.

Deploying automatically the load generator in Google Cloud

In order to automatically deploy the load generator, we created a terraform config that takes mainly the frontend address, type of virtual machines to preserve, number of instances to create and other variables to provision machines for running it querying the provided frontend. Each virtual machine will run a separate instance targeting the website.

You will be able to deploy the load generator in two ways. The first way is by directly using terraform, and the other way is by using a bash script that we have provided.

The first way to deploy is by navigating to the **load-generator-deployment** directory in our repository. Execute the following commands:

```
$ terraform init
$ terraform apply
```

During the execution, you will be prompted to provide values for the frontend address and the number of compute instances. These are the only variables without default values. Input the required information when prompted.

Alternatively, we offer a *more automated deployment* option through a Bash script named "**05-boutique-deploy-load-generator.sh**", which is located in the **scripts** directory. This script automates the deployment by automatically extracting the IP address of your **"frontend-external"** service and initiating the load generator.

```
$ ./05-boutique-deploy-load-generator.sh {INSTANCE_COUNT} <USERS_COUNT>
```

Advanced steps

Monitoring the application and the infrastructure

In this section, we implemented three distinct monitoring setups for our GKE cluster:

Auto-pilot Monitoring Deployment:

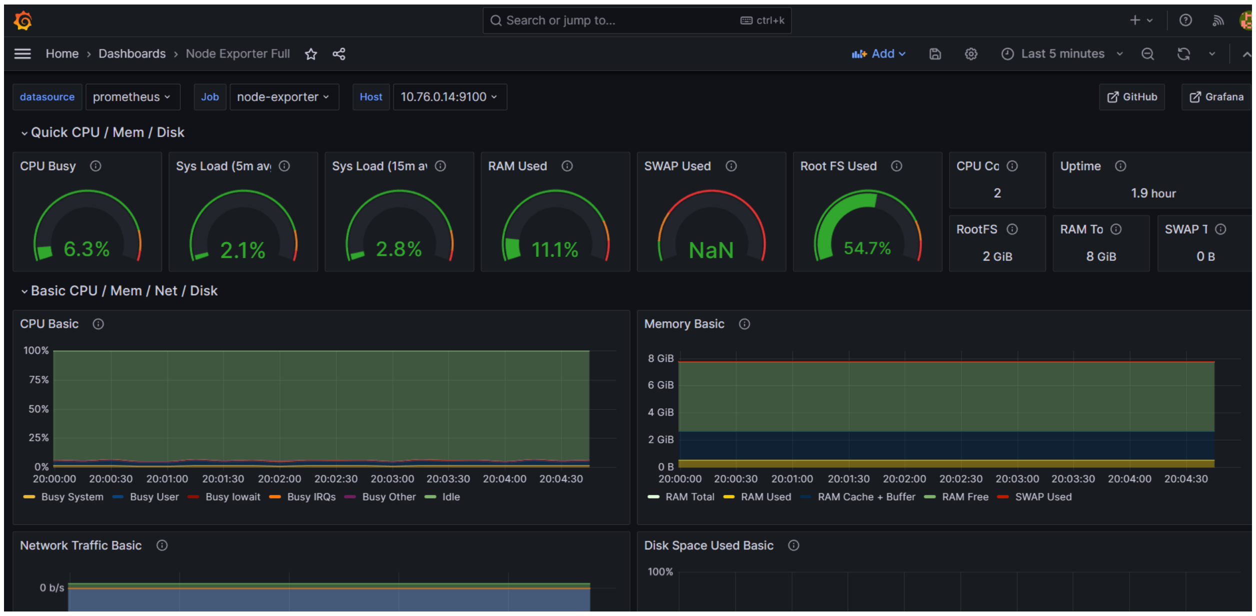
Initially, we attempted deploying the monitoring infrastructure in auto-pilot mode. However, we encountered a problem deploying the node exporter due to restrictions accessing a path in the node. After deploying Prometheus, we faced a 403 error while attempting to scrape the cadvisor endpoints. Granting maximum permissions and providing a “cluster-admin” role did not seem to suffice. We suspected that the kubelet configuration of the nodes required specific flags ('--authentication-token-webhook=true' and '--authorization-mode=Webhook') which we are not allowed to add in auto-pilot mode. The deployment in autopilot mode remains in our code, enabling monitoring setup without cadvisor functionality.

Standard Monitoring Deployment:

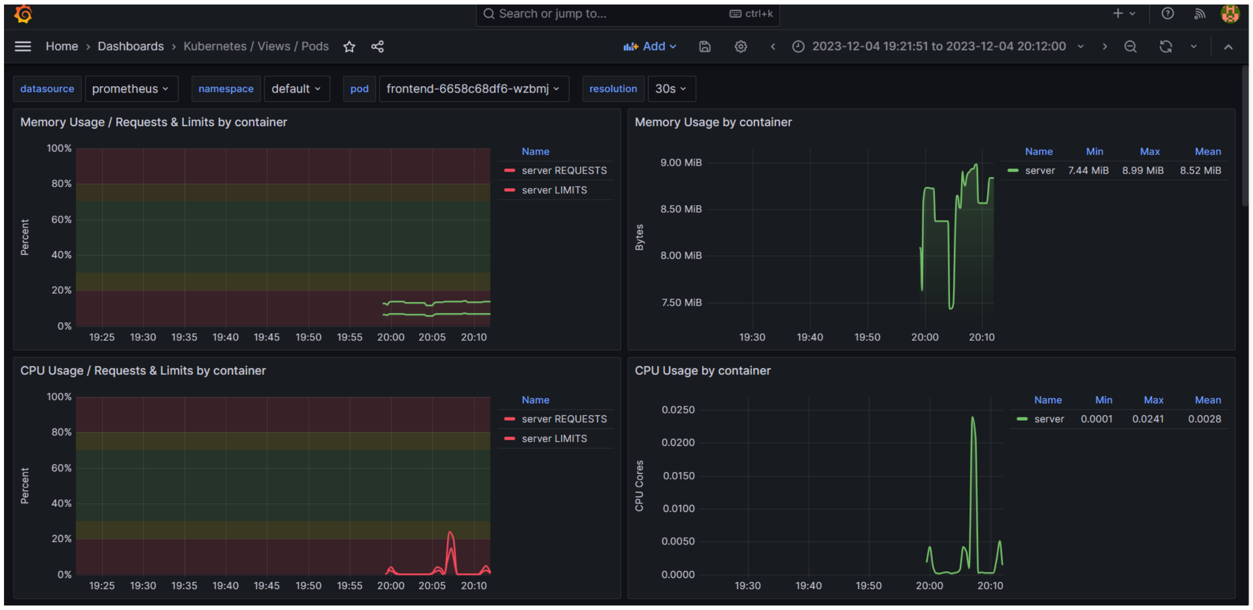
To address auto-pilot issues, we shifted to GKE standard deployment for better cluster control and permissions. By doing so, we successfully deployed Prometheus and Grafana along with node-exporter which gathers information about the nodes and kube-state-metrics which gathers information about the pods.

Furthermore, we were able to monitor resource consumption at both node and pod levels by importing specific dashboards:

For node-level information, the node exporter dashboard (**ID=1860**) can be imported.



To monitor pod-level resource consumption, the Kubernetes/views/pods dashboard (**ID=15760**) is available.


Search or jump to...
ctrl+k
Home › Dashboards › Kubernetes / Views / Pods
Add
2023-12-04 19:21:51 to 2023-12-04 20:12:00
datasource prometheus
namespace default
pod frontend-6658c68df6-wzbmj
resolution 30s
Memory Usage / Requests & Limits by container
Percent
Memory Usage by container
Bytes
CPU Usage / Requests & Limits by container
CPU Usage by container
CPU Cores
server REQUESTS
server LIMITS

Memory Usage by container

Name	Min	Max	Mean
server	7.44 MiB	8.99 MiB	8.52 MiB

CPU Usage by container

Name	Min	Max	Mean
server	0.0001	0.0241	0.0028

Standard Kube-Prometheus Repo Deployment:

Additionally, we included a deployment from the kube-prometheus repo. This encompassed:

- Prometheus Operator
- Prometheus
- Alertmanager
- Prometheus node-exporter
- Prometheus Adapter for Kubernetes Metrics APIs
- Kube-state-metrics
- Grafana

This deployment also brought preconfigured dashboards and crucial alerts for monitoring purposes.

For the first two setups, all the configs were provided from the following sources. These sources provided a step by step tutorial explaining what these services do and how they are deployed. We performed some modifications in order to set the service type as a Load Balancer, so we could have external access to their webpages.

- **Prometheus Setup Tutorial**
- **Node Exporter Setup Tutorial**
- **Grafana Setup Tutorial**
- **Kube-State-Metrics Setup Tutorial**

Deployment

For the streamlined deployment of all monitoring services, a provided bash script, namely **06-deploy-monitoring.sh** located in the **/scripts/** directory, has been made available. To initiate the deployment process, ensure that you have set the necessary permissions to execute the script. Once the permissions are configured, execute the script to deploy Prometheus, Node Exporters, and Grafana. This script simplifies the deployment steps, making it a straightforward and efficient process.

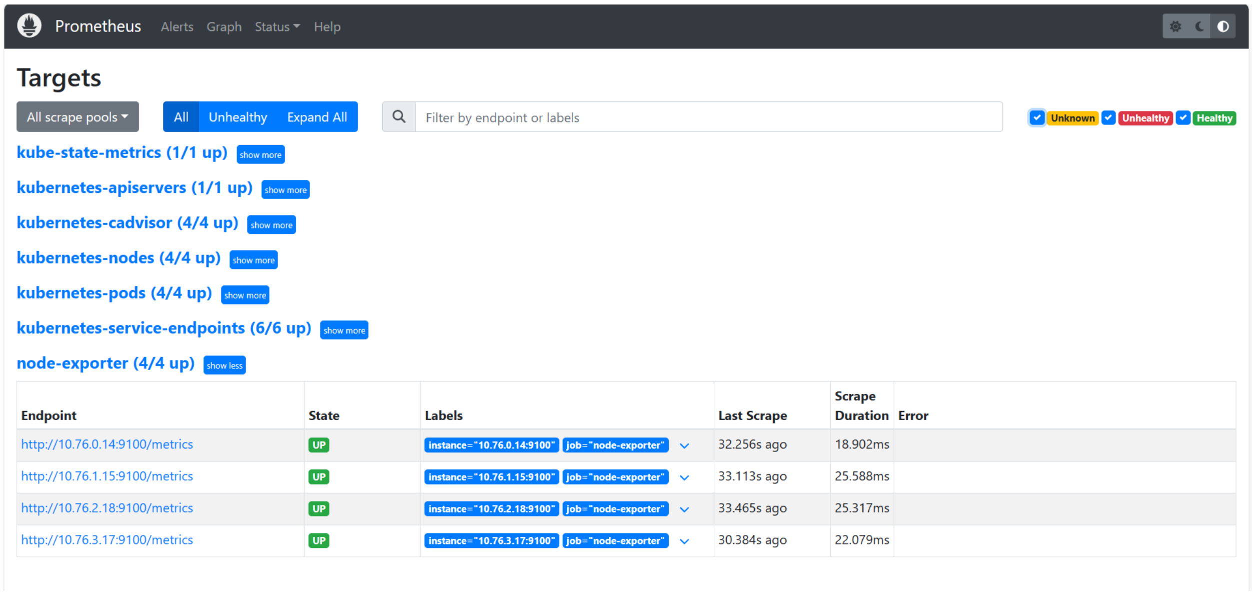
```
$ ./06-deploy-monitoring.sh
```

The script, by default, will use the files dedicated to the standard deployment. However, in case you opt to deploy in autopilot mode, it is possible to easily switch by modifying the DEPLOYMENT_TYPE in the script. Moreover, if you want to use the repository that contains the preconfigured dashboards and tools, as mentioned above, you can deploy that as well.

```
#DEPLOYMENT_TYPE="autopilot-deployment"
DEPLOYMENT_TYPE="standard-deployment"
#DEPLOYMENT_TYPE="standard-kube-prometheus-deployment"
```

Issues Faced & Solved

We encountered a challenge with Prometheus as we were unable to discover meaningful metrics. This arose from the absence of data being transferred to Prometheus. To address this issue, we found it necessary to incorporate the Node Exporter into our system.



Targets

kube-state-metrics (1/1 up) show more

kubernetes-apiservers (1/1 up) show more

kubernetes-cadvisor (4/4 up) show more

kubernetes-nodes (4/4 up) show more

kubernetes-pods (4/4 up) show more

kubernetes-service-endpoints (6/6 up) show more

node-exporter (4/4 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.76.0.14:9100/metrics	UP	instance="10.76.0.14:9100" job="node-exporter"	32.256s ago	18.902ms	
http://10.76.1.15:9100/metrics	UP	instance="10.76.1.15:9100" job="node-exporter"	33.113s ago	25.588ms	
http://10.76.2.18:9100/metrics	UP	instance="10.76.2.18:9100" job="node-exporter"	33.465s ago	25.317ms	
http://10.76.3.17:9100/metrics	UP	instance="10.76.3.17:9100" job="node-exporter"	30.384s ago	22.079ms	

Performance evaluation

In order to conduct a comprehensive performance evaluation reflecting real-life scenarios, we chose to leverage compute instances with higher resource capacities. While the initial plan was to deploy multiple basic instances, we encountered limitations on the number of available IP addresses that we could use. Therefore, we proceed to deploy instances with more resources to ensure a more robust and representative performance assessment. This shift allowed us to overcome constraints and better simulate real world scenarios.

To obtain data from Locust, we modified the bash script we use to initialize the instances' environments to export data stored in csv files. Moreover, we utilized the visualized data from both Grafana and Google Cloud Monitoring to further study and analyze the system's behavior.

Experimentation Process

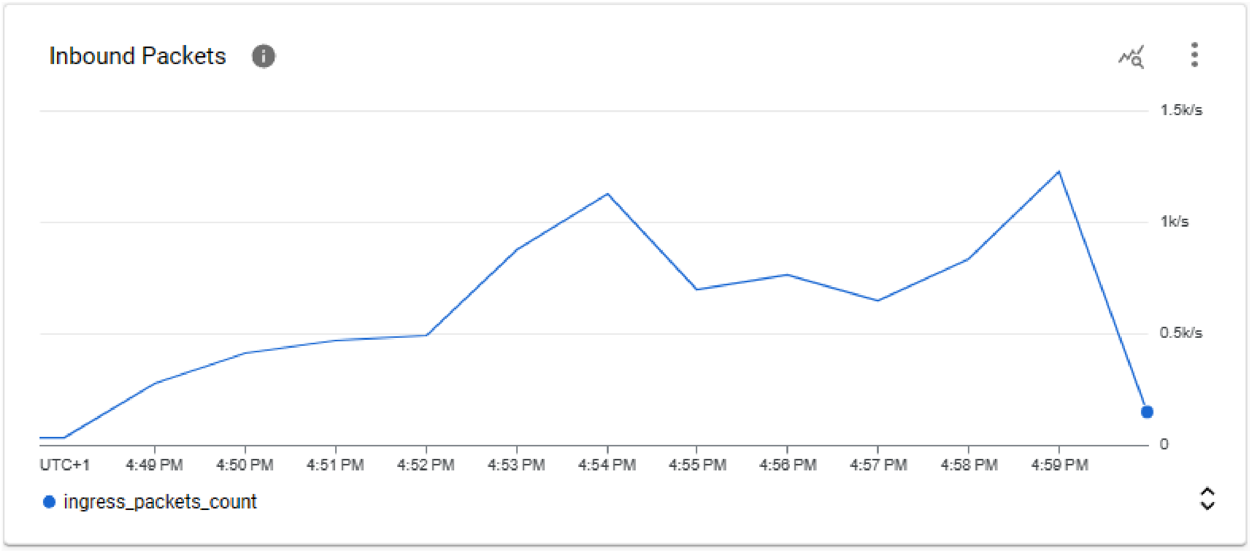
As previously mentioned, we opted for instances with more resources to simulate a scenario where our services experience a burst of traffic.

Specifications

- E2-standard-4
- Six Compute Instances
- 500 Users per Instance

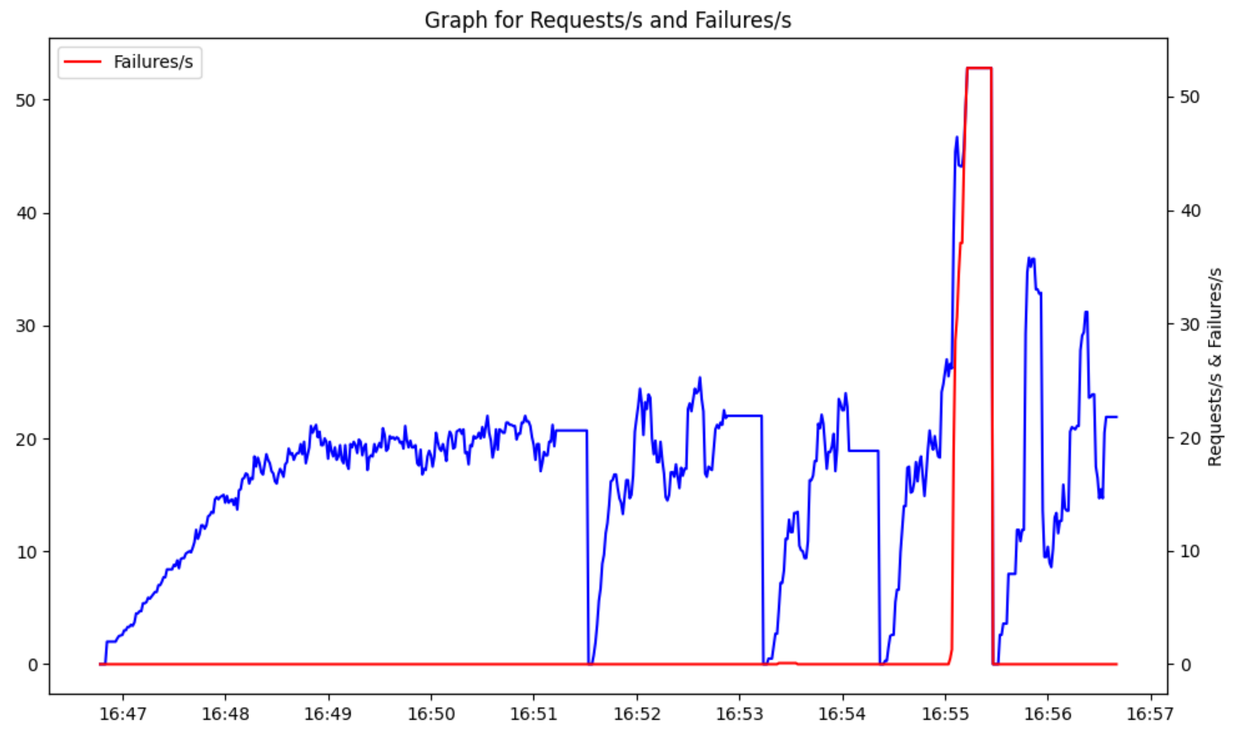
The implemented bash script introduces a deliberate 600-second (6 minutes) delay for each instance. This waiting period is designed to facilitate the collection of a substantial volume of meaningful data, allowing sufficient time for all users to submit their requests. Following this waiting period, the script proceeds to export the results to the **/var/log** directory.

Results & Analysis



The above graph depicts the inbound packets collected by Google Monitoring. The analysis of inbound packets provides valuable insights into the behavior of your system. Here's a breakdown of the observed patterns:

1. At the beginning, there is a slow growth in inbound packets. This corresponds to the initialization phase when all instances start running. During this time, the number of users and requests gradually increases.
2. At approximately 4:52 PM, a sudden spike in inbound packets is noted. This spike is attributed to a large influx of requests being sent simultaneously. This could be indicative of a peak usage period, due to increased user activity.
3. Around 4:54 PM, there is a sudden decrease in inbound packets. This abrupt drop is attributed to the failure of the pods responsible for intercepting this traffic. The failure of these pods likely led to a disruption in the processing and handling of incoming requests, resulting in a notable decrease in inbound packets.The decrease in inbound packets is caused by the fact that Locust operates synchronously. So whenever a user sends a request, this user won't send another request until the previous one is completed. Therefore, since the pod experienced a failure, and all the users are waiting for their requests to complete to proceed, no new requests will be sent, thus decreasing the number of inbound packets.
4. At around 4:57, we can see that the traffic increased again, which is due to the fact that the pods have restarted.



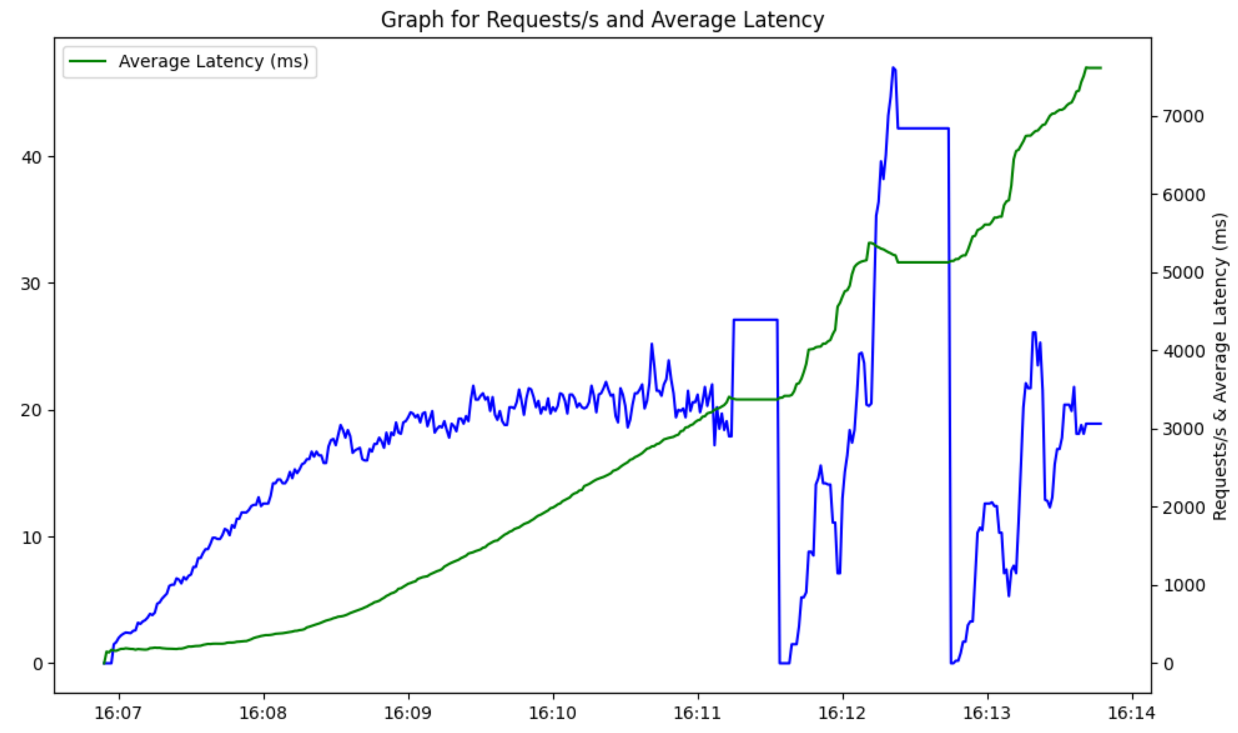
Examining the graph above, derived from Locust and illustrating the trend of Requests/Failures over time, parallels the patterns observed in inbound packet graphs. A gradual rise is evident during the initialization phase.

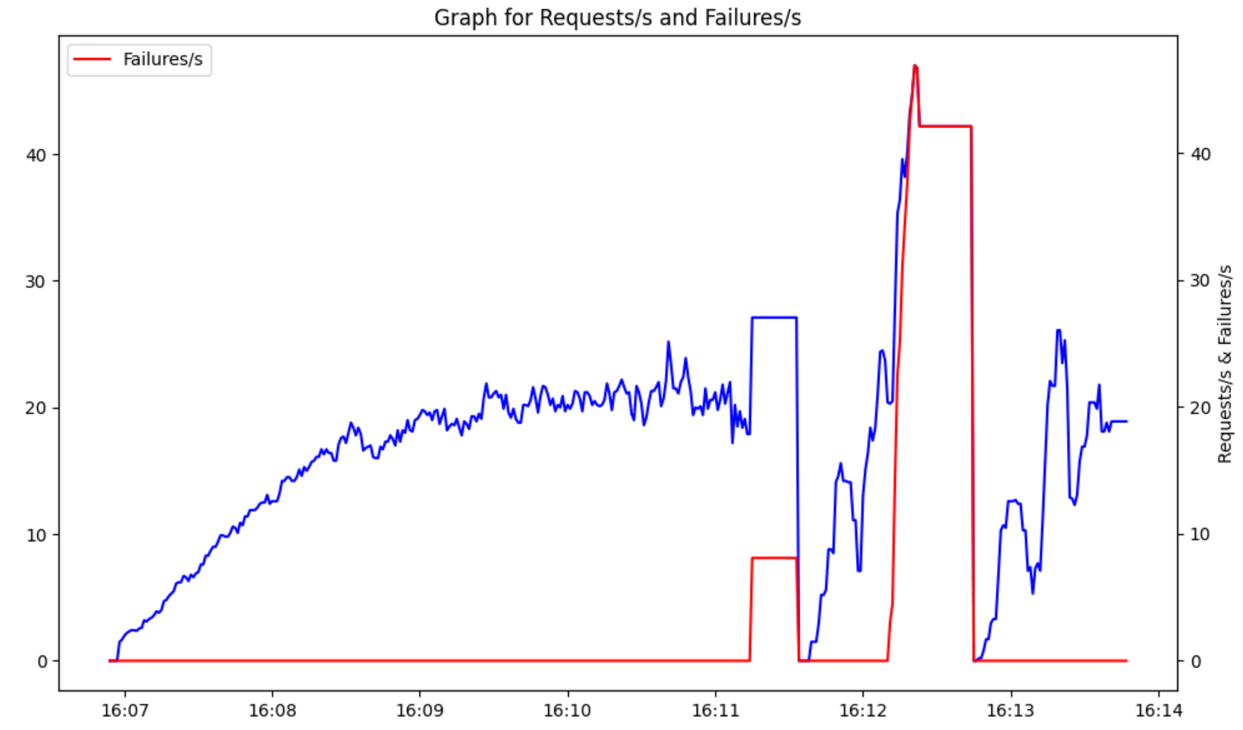
It's essential to note that Locust operates synchronously. When a user initiates a request, they await the completion of that specific request. Consequently, the graph exhibits instances where the number of requests stabilizes at a certain point, eventually dropping to 0. This behavior aligns with the synchronous nature of Locust, where periods of inactivity occur as users pause, waiting for the conclusion of their respective requests before initiating new ones. The visible stabilization points reflect these intervals of waiting between requests.

Around 16:45 (4:45 PM), mirroring the trend in the preceding graph, a sudden surge in requests per second is evident, followed by a rapid decline. Simultaneously, there is a notable increase in failures per second. As previously explained, this spike in failures can be directly attributed to the crash of pods or them reaching their limit. Consequently, users found themselves waiting for responses, causing the requests per second to drop to zero. This, in turn, resulted in a decrease in the number of inbound packets, as observed in the earlier graph.

This interconnected relationship between the spikes in requests, failures, and subsequent decreases underscores the impact of pod crashes on the overall system performance.

In another experiment that we had performed, we obtained the following graphs.

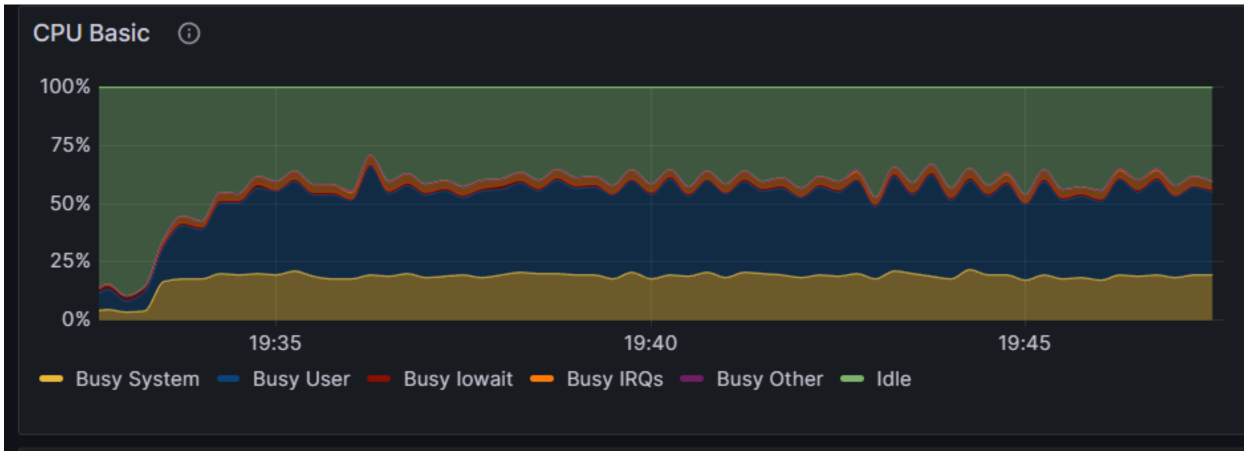




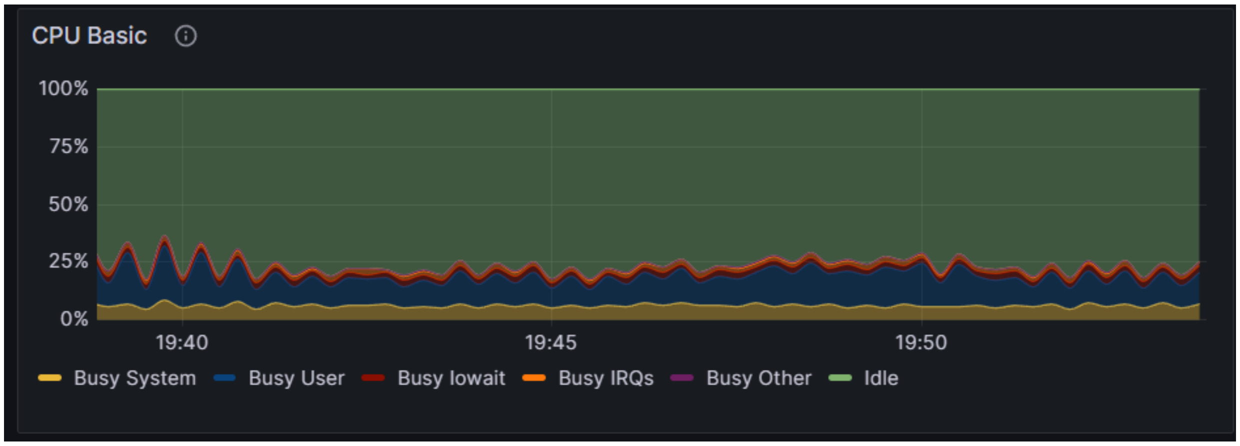
In the initial graph above, the upward trend in latency illustrates a consistent increase over time. This escalation is attributed to the continuous influx of requests to the service. As the volume of requests grows, the response time extends, reflecting the added load on the system.

In the subsequent graph, a notable inverse relationship between requests and failures is evident. However, it's crucial to note that a reduction in the number of requests doesn't necessarily guarantee a corresponding increase in failures. Nevertheless, in the event of failures, we observe that there will be a reduction in the volume of requests dispatched.

We have also collected data using Grafana. The graph below displays the CPU usage of the node, in which the frontend and currency services reside.



The graph below displays the CPU usage of a different node where other services reside.

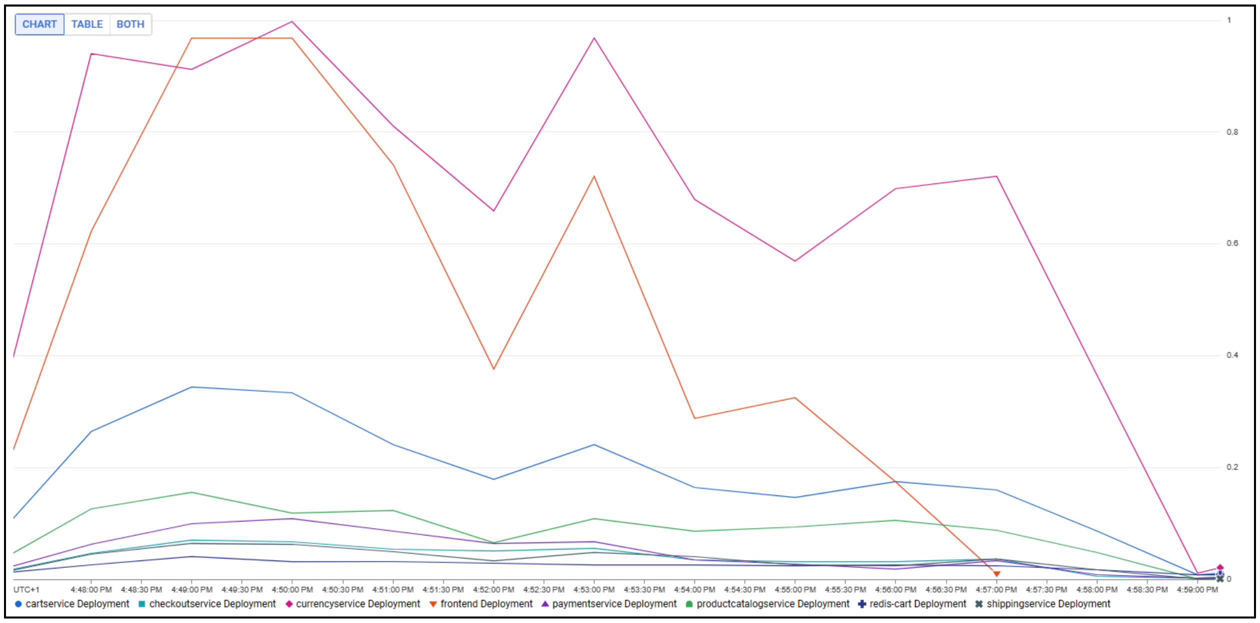


Upon examination of both graphical representations, a notable observation emerges. The node which has the frontend and currency services, as depicted in the first graph, demonstrates a high level of CPU utilization, indicating a frequent and intensive use. On the other hand, the second graph, portraying the CPU usage of the node where other services are deployed in, exhibit comparatively lower levels of CPU utilization when compared to the frontend and currency services.

This discrepancy suggests that while the frontend and currency services are consistently in demand, many other services on the second node, for example, are not being utilized as frequently. This insight could prompt further investigation into resource allocation, performance optimization, or potential dependencies impacting the overall efficiency of the system

Identifying bottlenecks [Bonus]

To identify the bottlenecks in the application, we deployed 6 e2-standard-4 machines with locust load generators and we monitored the resource usage of the pods. After a while, we observed that the currency and the frontend services were the ones reaching the CPU usage limit, and we noticed that failures from the clients occurred when they reached that limit, also the containers crashed and started restarting. This implies that the frontend service and the currency service are the bottlenecks of the application and should be scaled to scale the application.



Canary releases

We started by looking into the various ways of implementing the canary releases. We read about multiple ways of achieving it:

- It can be done by creating a new deployment and adjusting the number of replicas in such a way that the number of the new replicas of the new deployment is 25% of the total number of pods, but this is not efficient and complex to implement. First, because we can't easily have the percentages that we want. For example, if we want to have a canary release where 10% of the traffic going to the new version, then we will have to create 9 replicas of the old version and 1 replica of the new version. In addition to that, this would become much more complicated with pods autoscaling since the number of pods could change.
- We can automate the process before even GKE and having a google cloud deployment canary pipeline, but the problem of creating multiple replicas won't be solved.
- It can be also done using Istio and Flagger where we use Istio ingress gateway along with Flagger that is used for the deployment and analysis allowing automated rollbacks.

Approach

We used Kubernetes Gateway API with google cloud deployment to achieve the canary releases. For this part, we referred to **Google Cloud Canary Deploy Documentation**.
We employed the Istio gateway as the Kubernetes gateway for the frontend and established a delivery cloud pipeline for executing our canary deployment strategy. We structured the deployment in two distinct phases:

1. **Canary-25 Phase**: Redirects 25% of incoming requests to the new version while maintaining 75% of the traffic to the existing version.
2. **Stable Phase**: Shifts all incoming requests to the new version, and subsequently, the pods associated with the old version are systematically deleted.

Experimentation

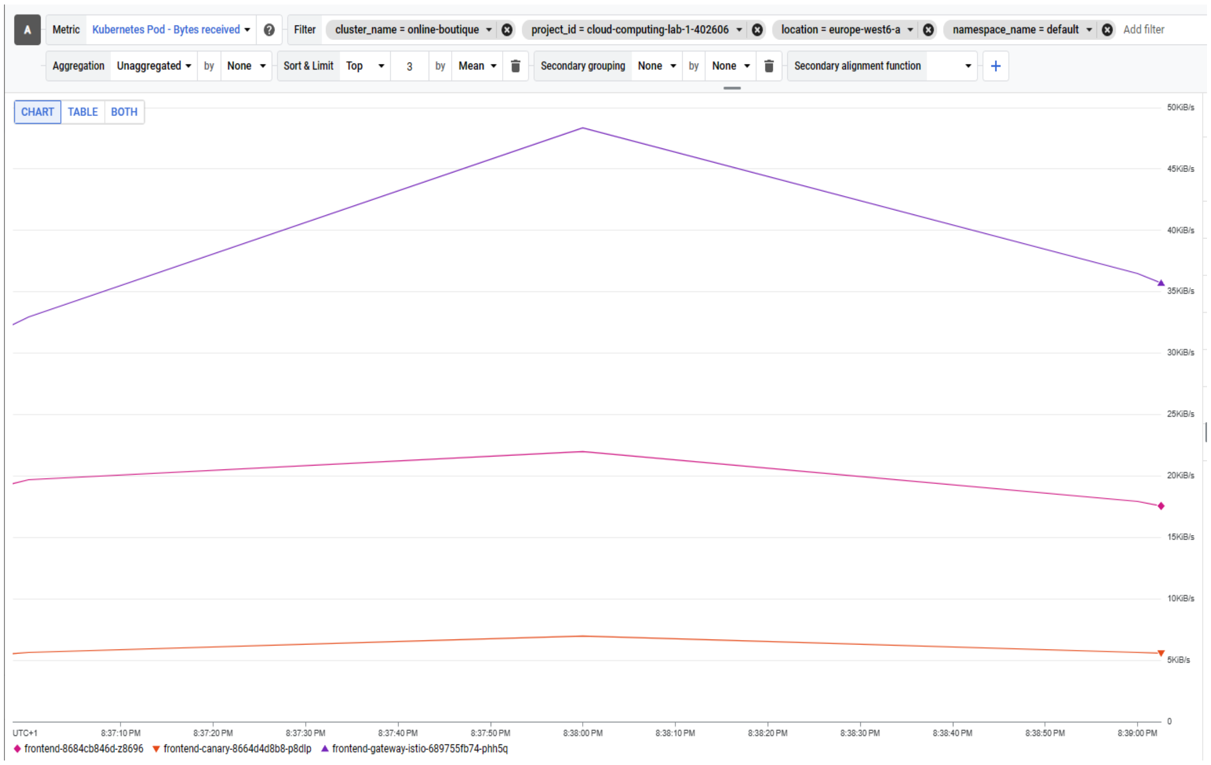
In our experimentation with releasing new versions, we made modifications to the frontend code by updating the text on the home page from "Hot Products" to "Hot Products V2." Following this, we built the image and pushed it to our Google Container Registry. To simplify the testing process for you, we've made the image publicly accessible, eliminating the need for you to build or push it. Concluding the process, we initiated a release using our Google Cloud Pipeline to seamlessly update the frontend with the latest changes.

Please note that the container registry will become deprecated in May.

Verification

There are numerous methods that we could follow to verify that the traffic splits with our canary release. The first method is to verify manually. For this method, we attempted to reload the website numerous times, and we observed that we are obtaining the new version around 25% of the time. However, this is not an efficient and useful method in real world applications, especially if our website receives a lot of traffic.

We also monitored the Pod's network as a second approach. We generated some load on the website and monitored the incoming traffic, which can be visualized using the graph below, on the pods that contain the old and new version. This method is a more realistic and useful approach, assuming that the request sizes are the same. However, this is also not the best approach to verify our deployment. After the canary release, we noticed that the pod with the new version is receiving about one third of the traffic, and we assumed that it is enough to verify.



Logs and metrics offer a highly practical method to confirm a canary release's accuracy. By emittings logs when a pod gets a request and associating the version of the service to them, we can create metrics based on these logs. These metrics would precisely track the number of requests received by each service version and would allow us to calculate the exact percentage received by each version. Unfortunately, due to time constraints, we weren't able to implement this.

Canary Releases [Bonus]

Multiple ways could be used to test and rollback a canary release, this could be possibly done with flagger and Istio allowing flagger to analyze the deployment based on metrics and rollback if necessary.

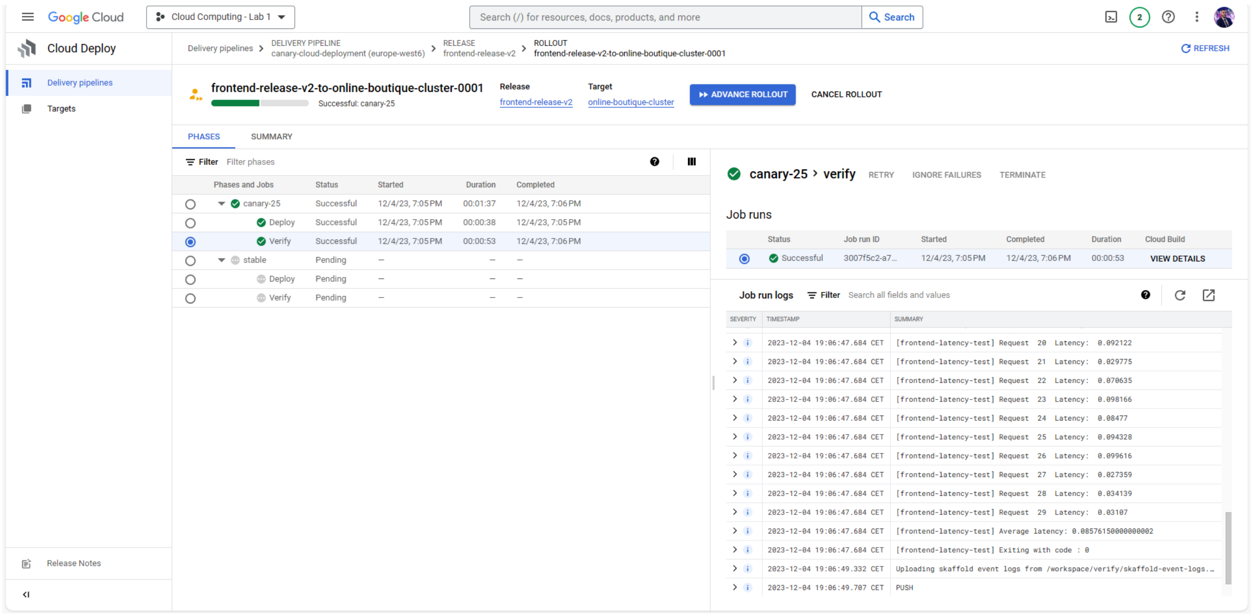
We created a new version of the frontend (v3) introducing 4 seconds of delay when visiting the homepage and we pushed the image publicly to Google Container Registry. To test the deployment, we added a verification test to Skaffold in the deployment pipeline and automated running it. So, whenever we run the canary deployment, the verification runs our tests and ensures they pass. The process is not fully automated for the rollback but it could be done.

We created one test for the latency. It is called frontend-latency-test. We configured the test to **run from within the kubernetes cluster**, it is a python docker image that send requests to the homepage of the website through the **canary frontend service only (not through the frontend gateway)**, it sends 30 requests, with a 1 second delay between each request, and

calculates the average latency. The test passes if the average latency is greater than 1.5 seconds.
[Verify your deployment | Cloud Deploy | Google Cloud]

As we can see below, we deployed the canary release of the frontend, the deployment succeeded and the verify test failed. However, it failed when deploying v3.

V2 Release



Delivery pipelines > DELIVERY PIPELINE canary-cloud-deployment (europe-west6) > RELEASE frontend-release-v2 > ROLLOUT frontend-release-v2-to-online-boutique-cluster-0001

frontend-release-v2-to-online-boutique-cluster-0001
Successful: canary-25
Release: frontend-release-v2
Target: online-boutique-cluster

Phases and Jobs	Status	Started	Duration	Completed
canary-25	Successful	12/4/23, 7:05 PM	00:01:37	12/4/23, 7:06 PM
Deploy	Successful	12/4/23, 7:05 PM	00:00:38	12/4/23, 7:05 PM
Verify	Successful	12/4/23, 7:05 PM	00:00:53	12/4/23, 7:06 PM
stable	Pending	–	–	–
Deploy	Pending	–	–	–
Verify	Pending	–	–	–

canary-25 > verify

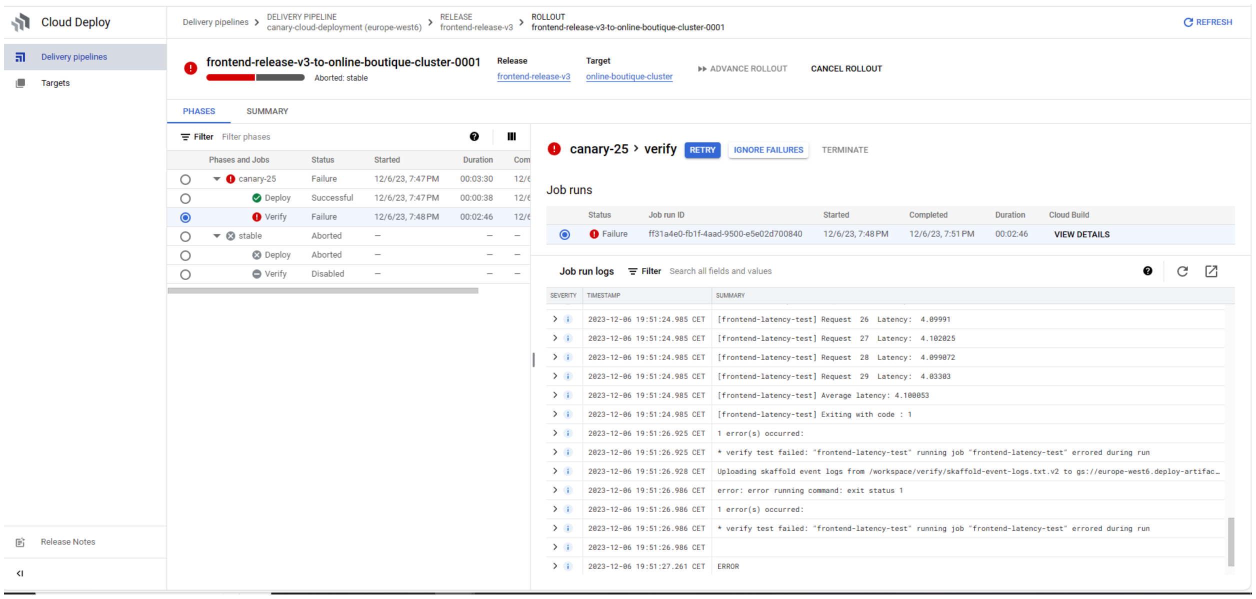
Job runs

Status	Job run ID	Started	Completed	Duration	Cloud Build
Successful	3007f5c2-a7…	12/4/23, 7:05 PM	12/4/23, 7:06 PM	00:00:53	VIEW DETAILS

Job run logs

```
SEVERITY  TIMESTAMP                    SUMMARY
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Request  20  Latency:  0.092122
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Request  21  Latency:  0.029775
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Request  22  Latency:  0.070635
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Request  23  Latency:  0.098166
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Request  24  Latency:  0.08477
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Request  25  Latency:  0.094328
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Request  26  Latency:  0.099616
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Request  27  Latency:  0.027359
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Request  28  Latency:  0.034139
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Request  29  Latency:  0.03107
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Average latency: 0.08576150000000002
i         2023-12-04 19:06:47.684 CET  [frontend-latency-test] Exiting with code : 0
i         2023-12-04 19:06:49.332 CET  Uploading skaffold event logs from /workspace/verify/skaffold-event-logs.…
i         2023-12-04 19:06:49.707 CET  PUSH
```

V3 Release



Delivery pipelines > DELIVERY PIPELINE canary-cloud-deployment (europe-west6) > RELEASE frontend-release-v3 > ROLLOUT frontend-release-v3-to-online-boutique-cluster-0001

frontend-release-v3-to-online-boutique-cluster-0001
Aborted: stable
Release: frontend-release-v3
Target: online-boutique-cluster

Phases and Jobs	Status	Started	Duration	Com
canary-25	Failure	12/6/23, 7:47 PM	00:03:30	12/6
Deploy	Successful	12/6/23, 7:47 PM	00:00:38	12/6
Verify	Failure	12/6/23, 7:48 PM	00:02:46	12/6
stable	Aborted	–	–	–
Deploy	Aborted	–	–	–
Verify	Disabled	–	–	–

canary-25 > verify

Job runs

Status	Job run ID	Started	Completed	Duration	Cloud Build
Failure	ff31a4e0-fb1f-4aad-9500-e5e02d700840	12/6/23, 7:48 PM	12/6/23, 7:51 PM	00:02:46	VIEW DETAILS

Job run logs

```
SEVERITY  TIMESTAMP                    SUMMARY
i         2023-12-06 19:51:24.985 CET  [frontend-latency-test] Request  26  Latency:  4.09991
i         2023-12-06 19:51:24.985 CET  [frontend-latency-test] Request  27  Latency:  4.102025
i         2023-12-06 19:51:24.985 CET  [frontend-latency-test] Request  28  Latency:  4.099072
i         2023-12-06 19:51:24.985 CET  [frontend-latency-test] Request  29  Latency:  4.03303
i         2023-12-06 19:51:24.985 CET  [frontend-latency-test] Average latency: 4.100053
i         2023-12-06 19:51:24.985 CET  [frontend-latency-test] Exiting with code : 1
i         2023-12-06 19:51:26.925 CET  1 error(s) occurred:
i         2023-12-06 19:51:26.925 CET  * verify test failed: "frontend-latency-test" running job "frontend-latency-test" errored during run
i         2023-12-06 19:51:26.928 CET  Uploading skaffold event logs from /workspace/verify/skaffold-event-logs.txt.v2 to gs://europe-west6.deploy-artifac…
i         2023-12-06 19:51:26.986 CET  error: error running command: exit status 1
i         2023-12-06 19:51:26.986 CET  1 error(s) occurred:
i         2023-12-06 19:51:26.986 CET  * verify test failed: "frontend-latency-test" running job "frontend-latency-test" errored during run
i         2023-12-06 19:51:26.986 CET
i         2023-12-06 19:51:27.261 CET  ERROR
```

Autoscaling [Bonus]

After evaluating performance and pinpointing the bottlenecks in the frontend and currency services, we suggest maintaining a minimum of 2 replicas at all times, especially since they occasionally crash. Horizontal auto scaling should be in place to adjust their numbers as needed. We can extend this autoscaling approach to other services, triggering horizontal auto scaling when CPU usage exceeds 70%.

Auto-pilot handled cluster scaling automatically by increasing node numbers. However, in standard mode, we strongly recommend configuring cluster autoscaling, considering its significance. Additionally, exploring vertical autoscaling with cautious limits from the outset could be beneficial. We could try scaling pods vertically to near zero and check the latency of our app and compare it to when we increase the resource limit.

Unfortunately, we didn't have sufficient time to implement and evaluate the auto scaling due to time constraints.

Managing Storage Backend for logging orders [Bonus]

Implementation

In this section, we opted for Python as the implementation language for the new microservice, leveraging our familiarity with the language. To facilitate the development and deployment process, we initiated the creation of a simple client-server application using gRPC. Our initial focus was on creating and ensuring a functional local server.

Subsequently, we integrated the Log Server's services into the proto file available in the microservices repository. This modified proto file enabled us to expose the Log Server's functions to other services, with a particular emphasis on the checkout service. The devised procedure for the checkout service involved the seamless addition of a new log by passing user and order IDs, along with the IDs of the items ordered.

To incorporate this procedure into the checkout service, we established a connection to the log server and made minor adjustments to the behavior of the checkout service to accommodate our new functionality. Importantly, we implemented this in a manner that ensures the checkout process remains uninterrupted in the event of a failure during the remote procedure call, promoting a smooth and reliable operation.

Deployment

In order to make the deployment workflow simpler, we opted for GitHub Actions to automate the image building and deployment processes. This allowed us to minimize manual intervention significantly. By configuring the necessary parameters, such as the image URL in the YAML file, GitHub Actions seamlessly handled the entire build and deploy pipeline.

To find the new/modified code for this section, please check out the **microservices** directory. For the log service, you only need to look at:

- Server.py

For the checkout service, you only need to look at:

- main.go on line 348 (excluding the lines of codes to establish a connection)

Deployment

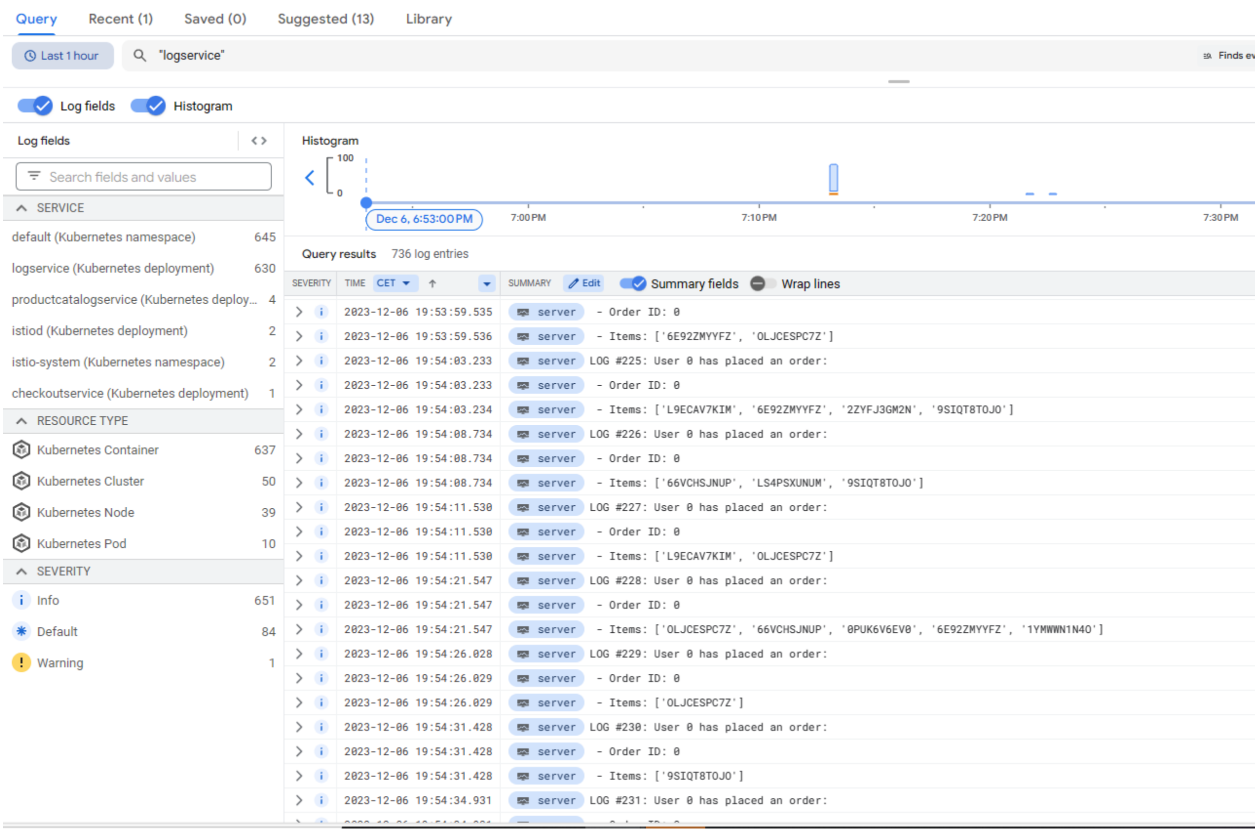
The service is automatically deployed when you first deploy the cluster.

Issues Faced and Solved

- Our implementation had the service storing the logs in a file. However, during deployment, the service didn't have the necessary permissions to write into a file. To solve this issue, we could either provide permissions or prevent the service from writing to a file; we chose the latter.
- Errors displaying while deploying the log service pod where the health checks failed. Although this might not be required to allow the service to run, it was included regardless.
- After modifying the checkout service, we encountered some crashes when deployed. In order to figure out the causes of these crashes, we compiled the service locally to determine what went wrong.

Using a storage backend for the log & persistence

Due to time constraints, we weren't able to implement a storage backend for the log service. Nevertheless, Google Cloud automatically stores our service logs by default. The implemented log service records order logs, and each of these logs is stored in **Cloud Logging**.



Log fields

SERVICE	
default (Kubernetes namespace)	645
logservice (Kubernetes deployment)	630
productcatalogservice (Kubernetes deploy…	4
istiod (Kubernetes deployment)	2
istio-system (Kubernetes namespace)	2
checkoutservice (Kubernetes deployment)	1
RESOURCE TYPE	
Kubernetes Container	637
Kubernetes Cluster	50
Kubernetes Node	39
Kubernetes Pod	10
SEVERITY	
Info	651
Default	84
Warning	1

Query results 736 log entries

```
SEVERITY  TIME CET                 SUMMARY
i  2023-12-06 19:53:59.535  server   - Order ID: 0
i  2023-12-06 19:53:59.536  server   - Items: ['6E92ZMYYFZ', 'OLJCESPC7Z']
i  2023-12-06 19:54:03.233  server  LOG #225: User 0 has placed an order:
i  2023-12-06 19:54:03.233  server   - Order ID: 0
i  2023-12-06 19:54:03.234  server   - Items: ['L9ECAV7KIM', '6E92ZMYYFZ', '2ZYFJ3GM2N', '9SIQT8TOJO']
i  2023-12-06 19:54:08.734  server  LOG #226: User 0 has placed an order:
i  2023-12-06 19:54:08.734  server   - Order ID: 0
i  2023-12-06 19:54:08.734  server   - Items: ['66VCHSJNUP', 'LS4PSXUNUM', '9SIQT8TOJO']
i  2023-12-06 19:54:11.530  server  LOG #227: User 0 has placed an order:
i  2023-12-06 19:54:11.530  server   - Order ID: 0
i  2023-12-06 19:54:11.530  server   - Items: ['L9ECAV7KIM', 'OLJCESPC7Z']
i  2023-12-06 19:54:21.547  server  LOG #228: User 0 has placed an order:
i  2023-12-06 19:54:21.547  server   - Order ID: 0
i  2023-12-06 19:54:21.547  server   - Items: ['OLJCESPC7Z', '66VCHSJNUP', '0PUK6V6EV0', '6E92ZMYYFZ', '1YMWWN1N40']
i  2023-12-06 19:54:26.028  server  LOG #229: User 0 has placed an order:
i  2023-12-06 19:54:26.029  server   - Order ID: 0
i  2023-12-06 19:54:26.029  server   - Items: ['OLJCESPC7Z']
i  2023-12-06 19:54:31.428  server  LOG #230: User 0 has placed an order:
i  2023-12-06 19:54:31.428  server   - Order ID: 0
i  2023-12-06 19:54:31.428  server   - Items: ['9SIQT8TOJO']
i  2023-12-06 19:54:34.931  server  LOG #231: User 0 has placed an order:
```

Despite this, our logs lack default structuring. By enhancing the log structure and integrating with Cloud Logging, we can achieve well-organized and meaningful logs.

Logs in Cloud Logging by Google are persistent [DOCS]. This means that they are stored in a dedicated, durable datastore and are not deleted automatically. By default, logs remain for a period of 30 days, with the ability to elongate that period. If we also want to store these for longer terms, we can export them to Cloud Storage