Scalable ML Model Deployment for Inference

Atchuth Naveen Chilaparasetti
achilapa@uci.edu

Hari Kishore Chaparala
hchapara@uci.edu

Manikanta Loya
manikanl@uci.edu

Introduction:

With the exponential growth of data, it has become increasingly important to develop applications leveraging insights drawn from data. Machine learning, in particular Deep learning algorithms, has proven effective in drawing inferences, especially in the field of image recognition, natural language processing, and predictive analytics. However, building and training deep learning models require significant computational resources and expertise. For example, the latest version of ChatGPT has 100 Trillion parameters! We also need a highly scalable and available infrastructure that can handle a huge volume of user queries over these ML models. Most such platforms use a high-end infrastructure or targeted fixed-scale deployments depending on the model they are hosting. Our platform, DL-Lambda is more generic and can support hosting multiple ML models and each ML model deployment can scale independently. Other approaches currently use Serverless architecture and while they may reduce the operating cost, they have a high startup time affecting availability and in some cases, the deployments are entirely managed by the cloud providers leading to low flexibility of the control plane by the host. In contrast, DL-Lamba is always available and we also add metric dashboards and fine-grained control for the host. DL-Lambda can be thought of as an alternative to a pure serverless architecture. In addition, we also support batching using queues leading to efficient resource utilization.

Project Goal & Design Formulation:

As part of this course project, we aim to design and implement a highly available and highly scalable distributed system that can offer Inference-As-A-Service. Our application will have two main components: 1) Web Interface, and 2) Backend to process the request. We use containerization and cloud services to scale and test our application.
In order to facilitate a high volume of uses we add a load balancer and horizontal scaling of resources. Then we decoupled our web interface and backend to optimize the resources for sporadic requests. Furthermore, to exploit batching and support multi-queries we added a queue connecting the web application and ML backend. To test our application in distributed deployment we use cloud services and evaluate metrics.

Design Choices:

The following are our design choices for each component:

Native vs Web Application: Unlike native applications, which require installation on the user's device, web applications do not need to be installed to be used. They can be accessed through any web browser, making them well-suited for simple applications. By implementing a server-client architecture - that is, running a web server on the cloud and fetching necessary data from it as needed - we can facilitate broader accessibility and cross-platform compatibility. Considering the above factors, we have designed our application to be a web application implemented in server-client architecture.

RabbitMQ vs Others: We considered message-oriented distributed platforms like Apache Kafka, Active MQ, and RabbitMQ for our middleware. Compared to Apache Kafka, which is inherently a pull based architecture, consumers in RabbitMQ(Push based approach) need not poll for the messages from the broker. This would avoid the additional overhead of pulling threads in client service. Also RabbitMQ implements newer AMQP(Advanced Multi Queueing Protocol) and supports other recent protocols like HTTP, STOMP, and MQTT, making it easier to integrate with other services. Compared to Active MQ(Java based middleware), RabbitMQ offers lower latency, and advanced routing capabilities. Moreover, for our use case, we don't require message persistence and streaming (offset resent and ordering) provided by Kafka. So Queue based platforms are a natural choice. Out of these three RabbitMQ provides easy-to-understand and straightforward APIs and is well-documented with examples.

Serverless vs Kubernetes: Although Serverless architecture seems to be a better approach to our event-driven application, it suffers from Cold-Start problems. The wait time for an initial user is high for a completely serverless application. With Kubernetes, we can guarantee the high availability by maintaining at least a single instance at all times. And the scaling can be triggered based on resource usage thereby avoiding Cold-Start problems.

GCP vs other cloud platforms vs multi-machine deployment: Kubernetes was originally developed by Google and although other cloud providers also offer Kubernetes support (AWS-EKS, Azure- AKS), GKE offered by GCP is the most up-to-date and optimized. Also, our ML models are trained using Google frameworks(TensorFlow) making it easier to integrate them in GCP. So we decided to deploy DL-Lambda in the Google Ecosystem. Moreover, GCP offers a $400 student credit with which we can use most of its services. AWS and Azure also offer some credit (~$100) but for the free tier, the number of usable services including Kubernetes is limited.

Frameworks Used:

1. **Docker:** Docker is a containerization platform that simplifies application packaging, distribution, and execution. It enables the creation and management of lightweight, isolated containers, ensuring consistent and reliable application deployment across different environments. Docker optimizes resource utilization, supports scalability, and facilitates versioning and rollbacks for efficient application management. It provides networking capabilities for seamless communication between containers and external networks. Integration with orchestration tools allows for easy scaling and load balancing.

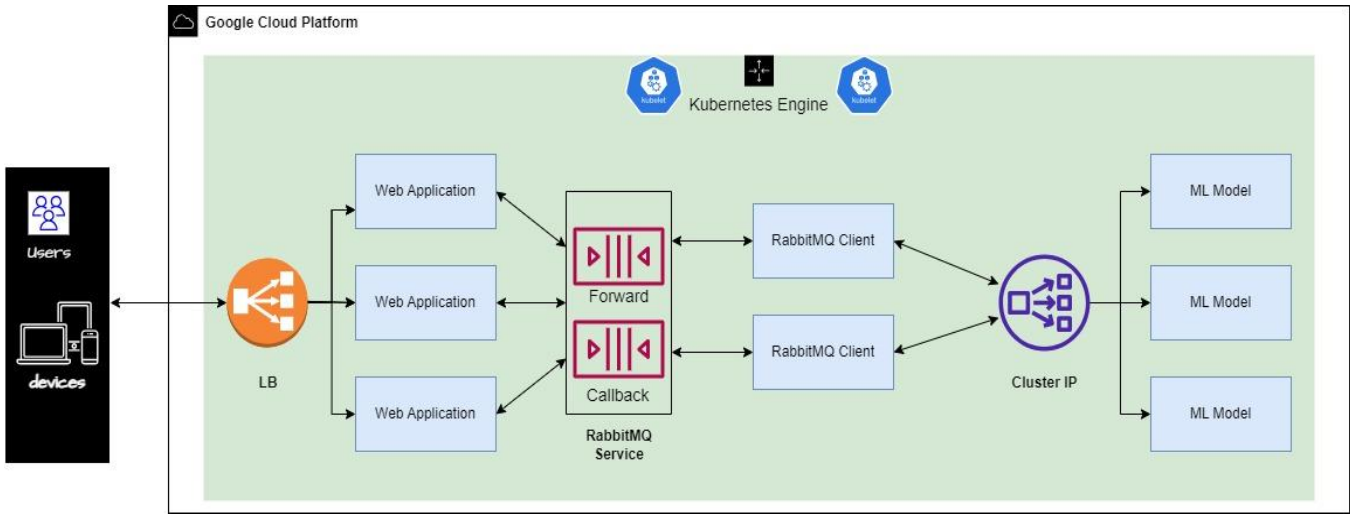
Docker's ecosystem and community offer a wide range of pre-built images, a marketplace, and community support for enhanced productivity. In summary, Docker revolutionizes the development and deployment of applications by providing a portable, efficient, and scalable containerization solution.

2. **Kuberbetes:** Kubernetes is a container orchestration platform that simplifies the management of containerized applications. It offers functionalities such as container management, scalability, self-healing, service discovery, storage orchestration, configuration management, rolling updates, resource allocation, multi-cloud support, and extensibility. Kubernetes automates container deployment, scaling, and scheduling, ensuring efficient resource utilization. It supports horizontal scaling and load balancing for high availability. Kubernetes monitors container health and performs automatic restarts or replacements for self-healing. Service discovery and networking enable seamless communication between containers. It facilitates storage provisioning and management. Kubernetes allows for configuration and secrets management, ensuring secure deployment. Rolling updates and rollbacks provide zero-downtime application updates. Resource allocation and monitoring enable efficient resource usage and troubleshooting. Kubernetes is cloud-agnostic, supporting multi-cloud and hybrid cloud environments. It has a rich ecosystem with plugins and extensions for enhanced functionality. In summary, Kubernetes empowers organizations to efficiently manage and scale containerized applications in dynamic and resilient environments.
3. **<u>Kubernetes pods:</u>** Kubernetes pods are the basic building blocks of an application deployment in a Kubernetes cluster. A pod is a group of one or more containers that are scheduled together on the same node and share the same network namespace. Containers within a pod can communicate with each other using localhost.Key features include
   Atomic Unit: A pod is the smallest and most basic deployable unit in Kubernetes. It represents a single instance of a process running on a cluster.
   Containers: Pods can contain one or more containers, typically running within a Docker or another container runtime. These containers are tightly coupled and share resources such as storage volumes, IP address, and port space.
   Shared Context: Containers within a pod share the same network namespace, which means they can communicate with each other using localhost. They can also share the same IPC namespace, allowing for inter-process communication.
   Pod Lifecycle: Pods have a lifecycle managed by the Kubernetes control plane. The control plane ensures that the desired number of pod replicas are running and restarts failed pods. Pods can be created, deleted, and scaled up or down based on workload demands.
4. **RabbitMQ:** RabbitMQ is a powerful open-source message broker that facilitates reliable and scalable messaging in distributed systems. It enables applications to communicate asynchronously and exchange data efficiently. With message queuing, RabbitMQ allows applications to send and receive messages in a decoupled manner, ensuring efficient communication between components. It follows a publish/subscribe model, enabling flexible message routing based on topic or content.

Message routing in RabbitMQ is achieved through various mechanisms, directing messages from exchanges to queues based on routing keys or patterns. Message acknowledgment ensures reliable delivery, as consumers can explicitly confirm the receipt and processing of messages.
RabbitMQ supports message persistence by storing messages on disk, preventing data loss even in the event of system failures. It also incorporates load-balancing capabilities, distributing messages across multiple consumers for optimal resource utilization.
Clustering enables fault tolerance and scalability by allowing multiple RabbitMQ nodes to form a single logical broker. Message transformation plugins facilitate data manipulation and integration with external systems during routing.
RabbitMQ provides management tools for easy administration and monitoring, offering insights into message rates, queue lengths, and system health. It also offers extensibility with client libraries for various programming languages, simplifying integration with different application environments.

5. **Tensorflow Serving:** TensorFlow Serving is a dedicated system for serving machine learning models built with TensorFlow. It supports model versioning, high-performance serving, flexible deployment, model monitoring, scalability, model security, and extensibility. It allows multiple versions of models to be served simultaneously, delivers low-latency and high-throughput serving, and offers various deployment options. TensorFlow Serving enables monitoring of model health and performance, handles high request volumes with load balancing, and provides security features for protected model access. It is designed to be customizable and extensible, allowing for tailored model serving pipelines and integration with additional functionalities. TensorFlow Serving is a reliable solution for efficiently serving TensorFlow models in production environments.
6. **Google Cloud Platform:** Google Cloud Platform (GCP) is a comprehensive suite of cloud computing services offered by Google. It includes virtual machines, containers, and serverless computing options for application deployment. GCP provides storage solutions, networking capabilities, and powerful tools for big data processing, analytics, and machine learning. It also offers security features, DevOps tools, and a global infrastructure for reliability and scalability. GCP enables businesses to build, deploy, and scale applications efficiently in the cloud, leveraging Google's expertise and resources.

System Design Diagram:



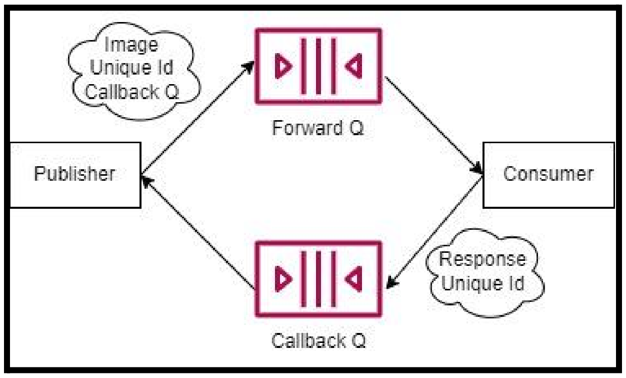
Application Workflow:

User Workflow:

A client can use our inference service by interacting with our web application through either API requests or web browser. As web applications are being run in Google Cloud Platform(GCP), users can access it from any web browser or device. Clients need to provide input(Images for this project) and a query that they need to infer(Currently, Is it a cat or not ?). The backend of the system then processes the request and replies back the answer. We use HTTP protocol for communication between users and web applications.

Application Middleware:

As soon as the user provides their query, these queries are then pushed to RabbitMQ. Here Web Applications acts as a publisher to RabbitMQ. In order to get the processed output back to the publisher for a remote consumer we implemented the RPC procedure with RabbitMQ. When a publisher publishes a message to the queue, it passes along a unique identifier for the message and information about the callback queue to which the consumer has to write the messages.



When RabbitClient(consumer) receives a message from a web application through RabbitMQ. It queries the ML Model server for further processing of images and inferences. Upon receiving a response from ML Model, it sends the response with a unique identifier along the callback queue it received as part of the query. We can use ‘routing keys’ to manage multiple queues and a variety of queries.

ML Model Server:

We use a pre-trained object detection computer vision model Resnet. Resnet is trained on the ImageNet dataset which contains 1000 classes. We use this model to predict whether the given image is a cat or dog. As part of the project we serve two different models one for cat detection and other for dog detection. Apart from these tasks we can have any number and variety of ML models as our backend. Each of these models host a web server to process the requests. And each of them can be containerized as an individual image. In addition, our ML model server also keeps updated with the latest model. We periodically fetch the latest model from the TensorFlow server. To prevent any downtime during the model update, we used a rolling deployment strategy where the old pods are deleted one by one as the new pods with the latest model update them.

Application Scalability:

All the components in DL-Lambda are individually scalable. We used the following strategy for autoscaling.

$$Window(1/podcount\ *\ \Sigma[pod_i\ CPU\ utilization\ ],\ 1\ min)\ >\ 0.8.$$

If the CPU utilization of pods with a 1 min window is greater than 80%, we add a new pod. And keep repeating the process.
For downscaling, we watch over a 10 min window to handle frequent traffic bursts.

$$Window(1/podcount\ *\ \Sigma[pod_i\ CPU\ utilization\ ],\ 10\ min)\ <\ 0.2$$

A new node (virtual machine) gets added if any pod cannot be provisioned.

Load Balancing:

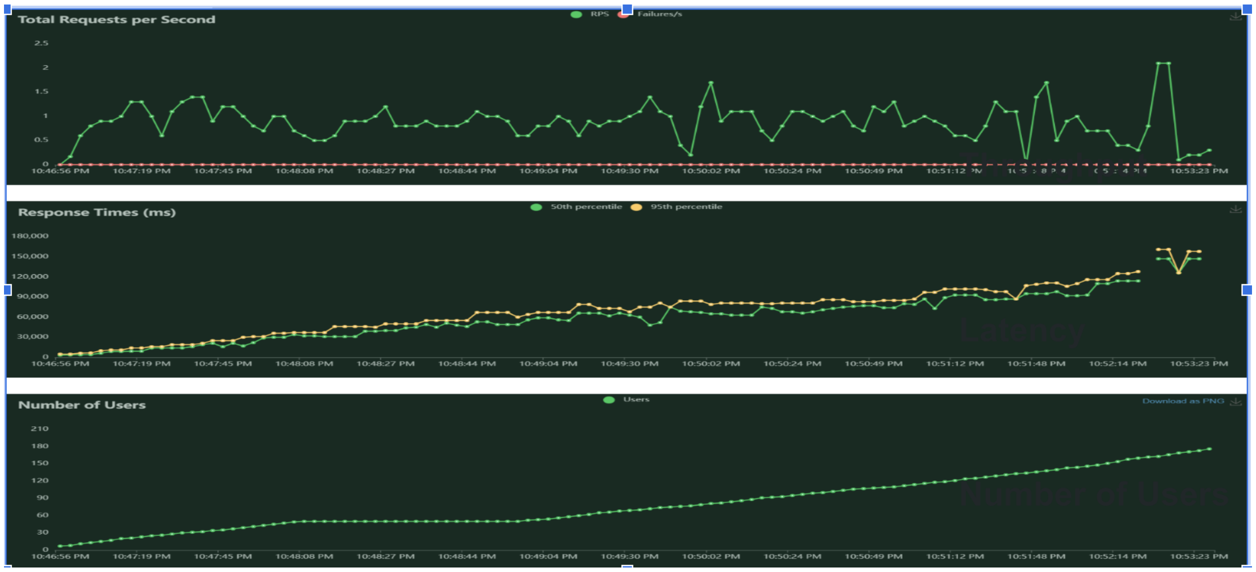
By default, Google offers load balancers that support hash-based and round-robin load balancing. As we wanted to minimize the latency, we opted for a different load-balancing option. Our approach also considers the CPU, Memory utilization of the pods to direct the traffic. If a pod is facing high resource utilization we send fewer requests to that pod.

Evaluation:

In this section, we see the evaluation of our distributed ML inference platform. First, we implemented our platform on a Kubernetes cluster on a single machine with a single node and then we evaluated the performance using the Locust Library.

Locust: Locust is an open-source, distributed, and scalable load-testing tool used to measure the performance and stability of software applications and systems. It allows developers and testers to simulate thousands or even millions of concurrent users to evaluate how an application handles high-traffic loads. Locust was created to address the need for an efficient and user-friendly load-testing solution that can simulate realistic user behavior. Locust is written in Python, which makes it easy to use and extend, and it leverages the power of the “gevent” library to handle massive concurrency efficiently. One of the key features of Locust is its ability to define user behavior using Python code, which provides great flexibility and control over load-testing scenarios. Users can define tasks that simulate various user actions, such as making HTTP requests, interacting with APIs, submitting forms, and parsing responses. This allows testers to create realistic and complex test scenarios that closely mimic the behavior of actual users. Another advantage of Locust is its web-based user interface, which provides real-time monitoring and reporting of the load test results. Testers can visualize key metrics like response times, request rates, and error rates through interactive charts and graphs. The user interface also allows for live changes to the test scenarios, enabling dynamic adjustments during the test execution.

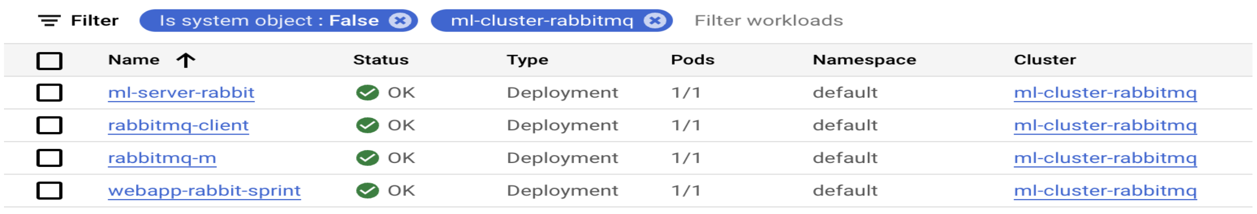
Below are the results generated from the locust on a single-machine deployment where the pods are using all the resources in the machine.



We see that with the increasing level of concurrency and number of requests in the third graph, the latency keeps increasing in the second graph, and the average response time increased from 30 sec to 150 seconds. The throughput also averages ~ 1 request per second.

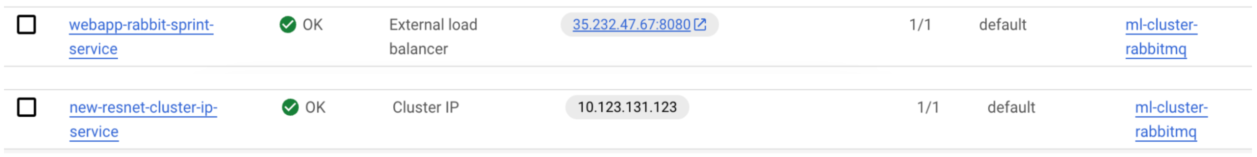
Next, we deploy our DL-Lambda in a multi-node environment in the Google Cloud platform. We have configured horizontal auto-scaling based on a service's aggregate CPU load. If the CPU load is > 80% for a small window we add a new pod to split the load. We haven’t configured pod vertical scaling as all these ML servers and web apps are handling a similar load. So the machine specification (CPUs & memory) is correctly set beforehand. We use a rollover deployment strategy that ensures that only one pod goes down at any given time. For fault tolerance, if any pod goes down, we have configured the deployment such that a new pod can immediately be created. For high availability, we based our Kubernetes nodes across several zones and regions to handle data center outages. In addition, we have added a sidecar container for metrics collection within our Kubernetes cluster.

The below image shows our original deployment which shows our deployments for services (ml-server, web app, RabbitMQ client, and RabbitMQ) all contain a single pod. For RabbitMQ deployment, we have chosen persistent storage and deployment which ensures that its always available unless a node failure happens. We have also set its resources to a high value (4GB and 4 Cores with 10GB persistent storage) and set the deployment to just one pod.

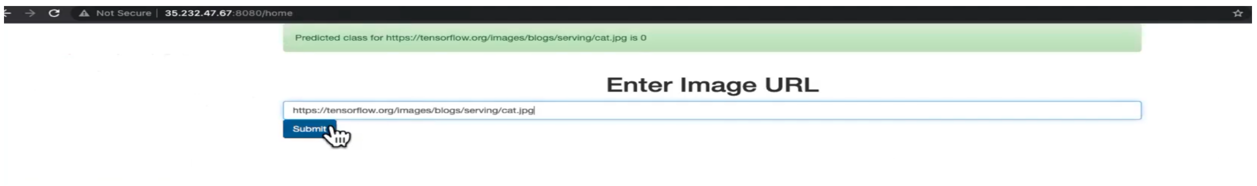


Name	Status	Type	Pods	Namespace	Cluster
ml-server-rabbit	OK	Deployment	1/1	default	ml-cluster-rabbitmq
rabbitmq-client	OK	Deployment	1/1	default	ml-cluster-rabbitmq
rabbitmq-m	OK	Deployment	1/1	default	ml-cluster-rabbitmq
webapp-rabbit-sprint	OK	Deployment	1/1	default	ml-cluster-rabbitmq

We also created an external load balancer for Web App and an internal ClusterIP for the ML server. The external IP address is public and it provides a web interface to interact with DL-Lambda



webapp-rabbit-sprint-service	OK	External load balancer	35.232.47.67:8080	1/1	default	ml-cluster-rabbitmq
new-resnet-cluster-ip-service	OK	Cluster IP	10.123.131.123	1/1	default	ml-cluster-rabbitmq

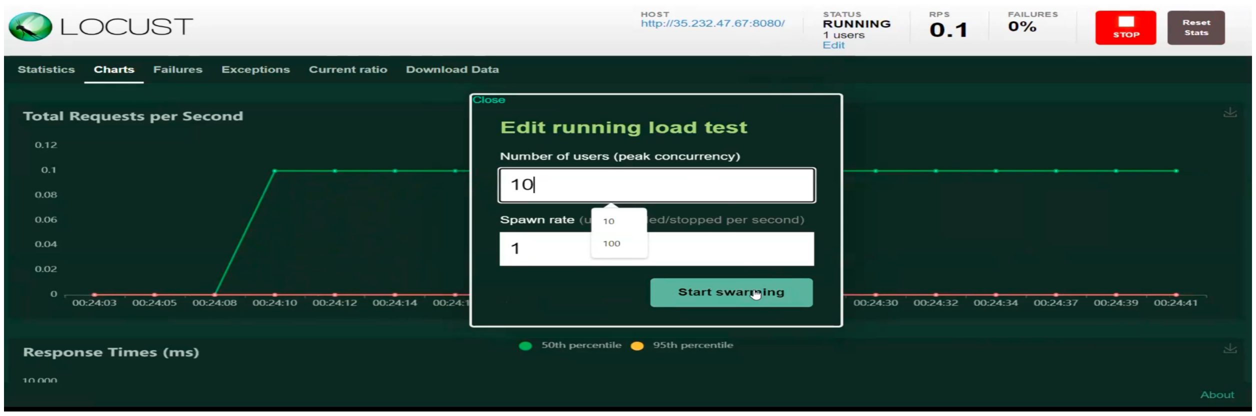


Predicted class for https://tensorflow.org/images/blogs/serving/cat.jpg is 0

Enter Image URL

https://tensorflow.org/images/blogs/serving/cat.jpg

Horizontal Autoscaling: Now, we trigger a locust load test to check DL-Lamba autoscaling.

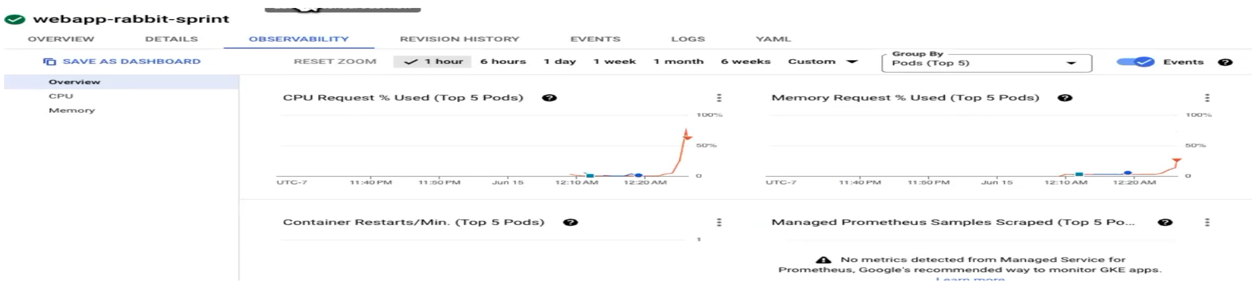


Edit running load test

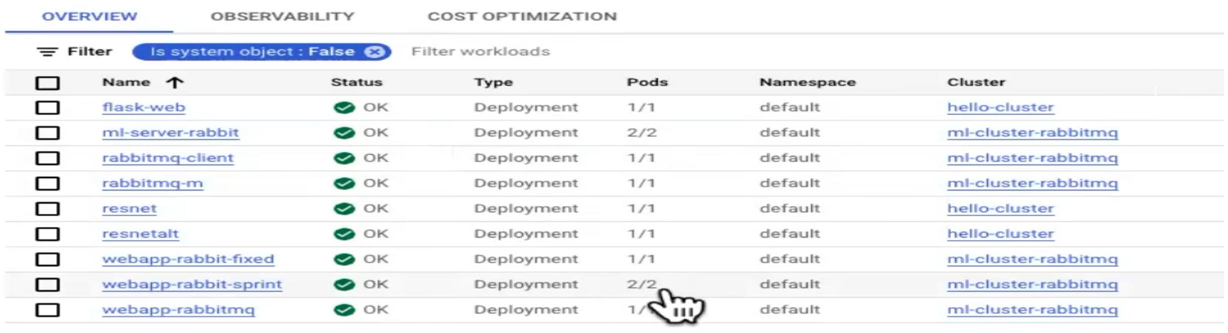
Number of users (peak concurrency): 10

Spawn rate (users started/stopped per second): 1

We see that for the web app, the CPU usage went above 80% and so our horizontal scaling will trigger creating the desired number of pods.

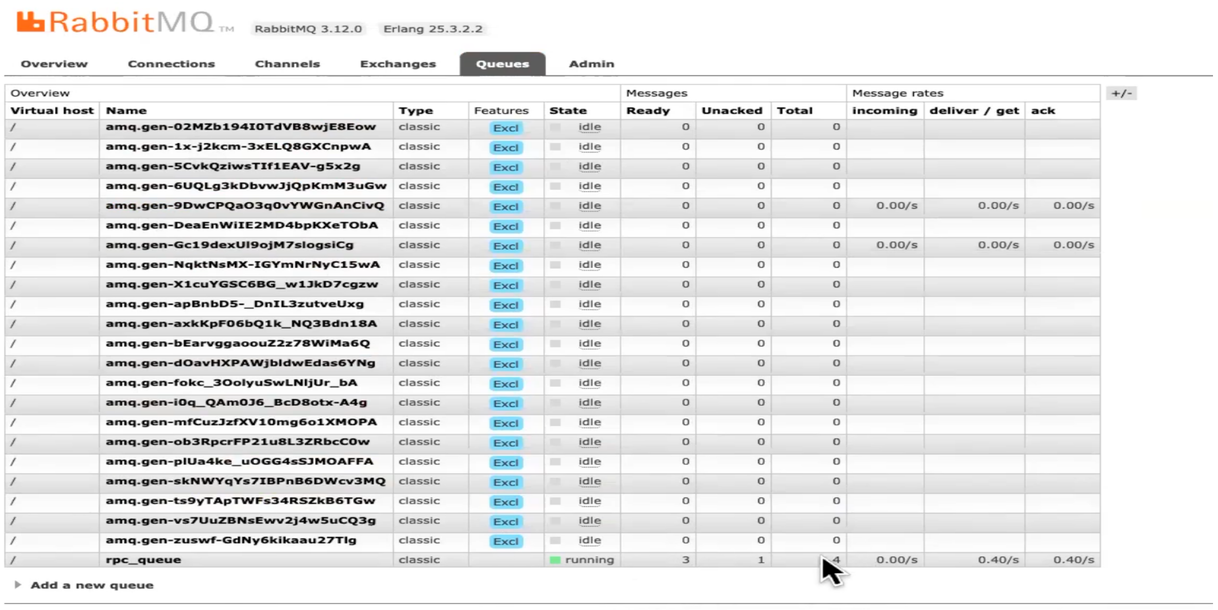

webapp-rabbit-sprint
OVERVIEW
DETAILS
OBSERVABILITY
REVISION HISTORY
EVENTS
LOGS
YAML
SAVE AS DASHBOARD
RESET ZOOM
1 hour
6 hours
1 day
1 week
1 month
6 weeks
Custom
Group By Pods (Top 5)
Events
Overview
CPU
Memory
CPU Request % Used (Top 5 Pods)
Memory Request % Used (Top 5 Pods)
Container Restarts/Min. (Top 5 Pods)
Managed Prometheus Samples Scraped (Top 5 Po...
No metrics detected from Managed Service for Prometheus, Google's recommended way to monitor GKE apps.

Below, we see that both the ml-server and web app are scaled to two pods now. Note that all our deployments are auto-scalable except the RabbitMQ instance.



Name	Status	Type	Pods	Namespace	Cluster
flask-web	OK	Deployment	1/1	default	hello-cluster
ml-server-rabbit	OK	Deployment	2/2	default	ml-cluster-rabbitmq
rabbitmq-client	OK	Deployment	1/1	default	ml-cluster-rabbitmq
rabbitmq-m	OK	Deployment	1/1	default	ml-cluster-rabbitmq
resnet	OK	Deployment	1/1	default	hello-cluster
resnetalt	OK	Deployment	1/1	default	hello-cluster
webapp-rabbit-fixed	OK	Deployment	1/1	default	ml-cluster-rabbitmq
webapp-rabbit-sprint	OK	Deployment	2/2	default	ml-cluster-rabbitmq
webapp-rabbitmq	OK	Deployment	1/	default	ml-cluster-rabbitmq

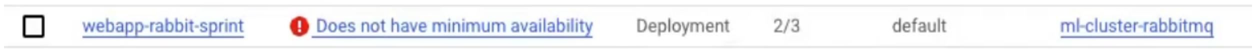
Our deployment also uses multiple queues for batching as we don't want a single queue as the bottleneck.



Overview					Messages			Message rates			+/-
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
/	amq.gen-02MZb194I0TdVB8wjE8Eow	classic	Excl	idle	0	0	0				
/	amq.gen-1x-j2kcm-3xELQ8GXCnpwA	classic	Excl	idle	0	0	0				
/	amq.gen-5CvkQziwsTIf1EAV-g5x2g	classic	Excl	idle	0	0	0				
/	amq.gen-6UQLg3kDbvwJjQpKmM3uGw	classic	Excl	idle	0	0	0				
/	amq.gen-9DwCPQaO3q0vYWGnAnCivQ	classic	Excl	idle	0	0	0	0.00/s	0.00/s	0.00/s	
/	amq.gen-DeaEnWiIE2MD4bpKXeTObA	classic	Excl	idle	0	0	0				
/	amq.gen-Gc19dexUl9ojM7slogsiCg	classic	Excl	idle	0	0	0	0.00/s	0.00/s	0.00/s	
/	amq.gen-NqktNsMX-IGYmNrNyC15wA	classic	Excl	idle	0	0	0				
/	amq.gen-X1cuYGSC6BG_w1JkD7cgzw	classic	Excl	idle	0	0	0				
/	amq.gen-apBnbD5-_DnIL3zutveUxg	classic	Excl	idle	0	0	0				
/	amq.gen-axkKpF06bQ1k_NQ3Bdn18A	classic	Excl	idle	0	0	0				
/	amq.gen-bEarvggaoouZ2z78WiMa6Q	classic	Excl	idle	0	0	0				
/	amq.gen-dOavHXPAWjbldwEdas6YNg	classic	Excl	idle	0	0	0				
/	amq.gen-fokc_3OolyuSwLNljUr_bA	classic	Excl	idle	0	0	0				
/	amq.gen-i0q_QAm0J6_BcD8otx-A4g	classic	Excl	idle	0	0	0				
/	amq.gen-mfCuzJzfXV10mg6o1XMOPA	classic	Excl	idle	0	0	0				
/	amq.gen-ob3RpcrFP21u8L3ZRbcC0w	classic	Excl	idle	0	0	0				
/	amq.gen-plUa4ke_uOGG4sSJMOAFFA	classic	Excl	idle	0	0	0				
/	amq.gen-skNWYqYs7IBPnB6DWcv3MQ	classic	Excl	idle	0	0	0				
/	amq.gen-ts9yTApTWFs34RSZkB6TGw	classic	Excl	idle	0	0	0				
/	amq.gen-vs7UuZBNsEwv2j4w5uCQ3g	classic	Excl	idle	0	0	0				
/	amq.gen-zuswf-GdNy6kikaau27Tlg	classic	Excl	idle	0	0	0				
/	rpc_queue	classic		🟩 running	3	1	4	0.00/s	0.40/s	0.40/s	

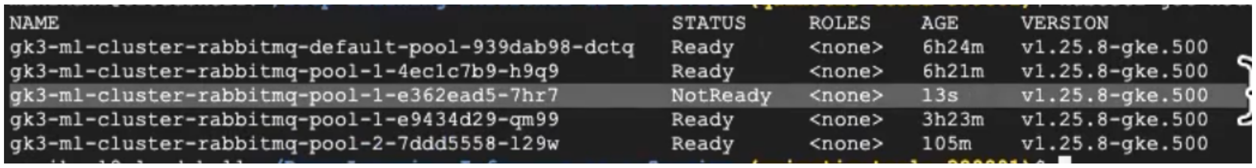
Add a new queue

Autoscaling when the node pool is out of resources for a new pod allocation: We further increase the load generated by the Locust but this time, we don't have enough resources (CPU and memory) for a new pod allocation as shown by the below error.

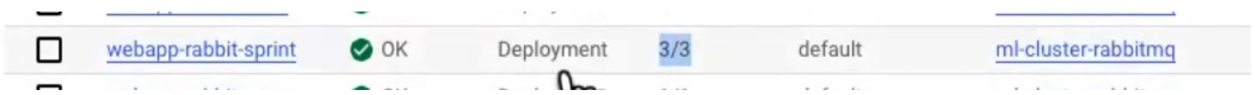


☐	webapp-rabbit-sprint	❗ Does not have minimum availability	Deployment	2/3	default	ml-cluster-rabbitmq

As we configured for a flexible node pool with node addition and removal, a new node now gets added to the pool to provision for the new pod.

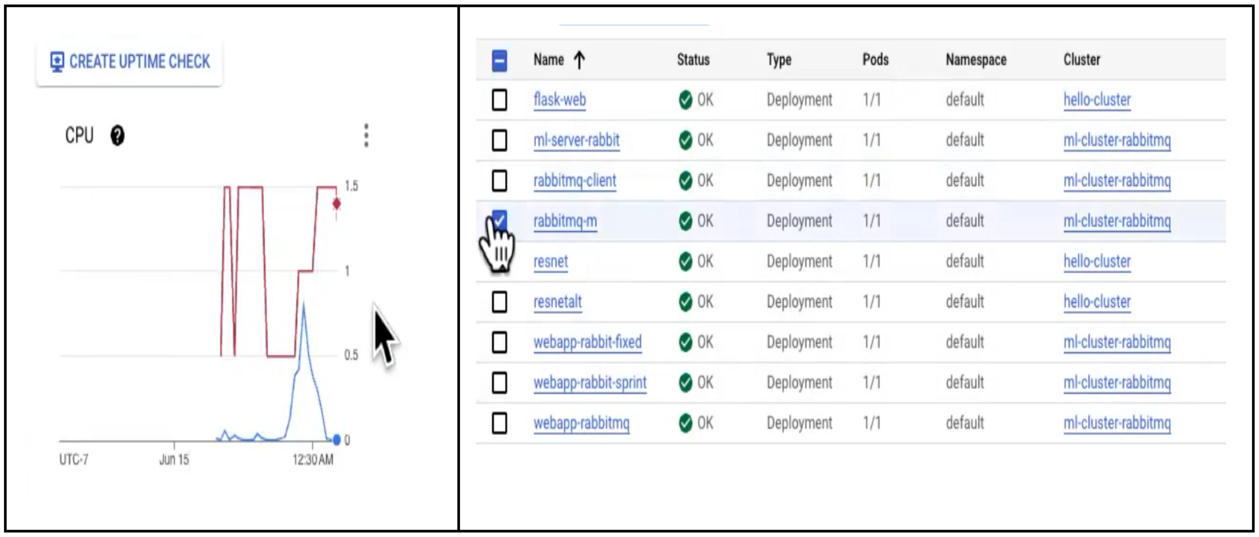


```
NAME                                              STATUS     ROLES    AGE     VERSION
gk3-ml-cluster-rabbitmq-default-pool-939dab98-dctq   Ready      <none>   6h24m   v1.25.8-gke.500
gk3-ml-cluster-rabbitmq-pool-1-4ec1c7b9-h9q9         Ready      <none>   6h21m   v1.25.8-gke.500
gk3-ml-cluster-rabbitmq-pool-1-e362ead5-7hr7         NotReady   <none>   13s     v1.25.8-gke.500
gk3-ml-cluster-rabbitmq-pool-1-e9434d29-qm99         Ready      <none>   3h23m   v1.25.8-gke.500
gk3-ml-cluster-rabbitmq-pool-2-7ddd5558-129w         Ready      <none>   105m    v1.25.8-gke.500
```

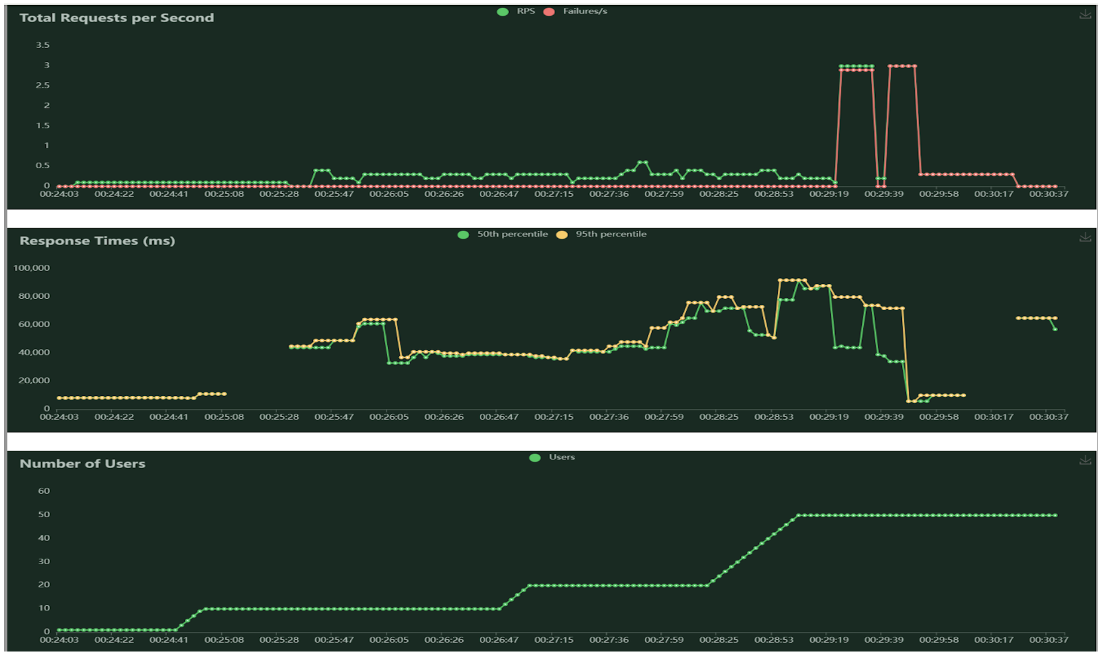


☐	webapp-rabbit-sprint	✓ OK	Deployment	3/3	default	ml-cluster-rabbitmq

Next, we stop the locust swarm and see that the load has decreased which should scale down the pods and node pool back to the original state. For upscaling, we configured it to be done using the last 1-minute window and for downscaled we used the last 10 min window to prevent any service downgrading in case of frequent bursts of traffic.



Name	Status	Type	Pods	Namespace	Cluster
flask-web	OK	Deployment	1/1	default	hello-cluster
ml-server-rabbit	OK	Deployment	1/1	default	ml-cluster-rabbitmq
rabbitmq-client	OK	Deployment	1/1	default	ml-cluster-rabbitmq
rabbitmq-m	OK	Deployment	1/1	default	ml-cluster-rabbitmq
resnet	OK	Deployment	1/1	default	hello-cluster
resnetalt	OK	Deployment	1/1	default	hello-cluster
webapp-rabbit-fixed	OK	Deployment	1/1	default	ml-cluster-rabbitmq
webapp-rabbit-sprint	OK	Deployment	1/1	default	ml-cluster-rabbitmq
webapp-rabbitmq	OK	Deployment	1/1	default	ml-cluster-rabbitmq



Locust metrics during Autoscaling:

Here we see that unlike with the single machine, the latency initially increases, and when the pods scale up, it goes down. And when the pods increase further the latency returns to the original value. The latency varied from 20 sec to 80 seconds averaging at ~ 50 sec which is a significant improvement compared to the single node deployment where the latency was ~ 150 seconds with the same load and also performed on the same machine. The throughput also keeps improvising with the number of pods, unlike the single node machine where it remained constant.

We also observe that there were some failures during autoscaling. This is due to the pods getting shuffled due to the newly created node. Since we are batching the user queries and one pod got allocated to a different node, there was a downtime of around 5 seconds and we see a ~50% failure rate. This can be prevented by fixing the pods similar to what we have done for RabbitMQ but since the downtime is low and we need to ensure fault tolerance, we have decided to not disable pod reallocation. The users can always retry the requests after a short downtime during node addition.

Applications:

The best applications for DL-Lambda are high-throughput real-time applications that can have varying traffic over time. Organizations can use DL-Lambda to deploy their own ML models are different Kubernetes deployments and can leverage its fault-tolerance, high availability, and auto-scaling capabilities. This allows for low-latency responses and cost-efficient deployment. Some sample applications include:

1. Real-time Image or Video Classification: DL-Lambda can be used to serve deep learning models for real-time image or video classification tasks. It can process incoming images or video frames and provide instant predictions, enabling applications such as object recognition, facial recognition, or video content analysis.

2. Natural Language Processing (NLP): DL-Lambda can be employed to serve NLP models for tasks like sentiment analysis, text classification, language translation, or chatbot responses (Similar to ChatGPT). It can handle large volumes of user queries and provide quick and accurate predictions.

3. Recommendation Systems: DL-Lambda can power recommendation systems by serving deep learning models that offer personalized recommendations to users based on their preferences and behavior. It can handle high query loads and provide real-time recommendations, enhancing user experience and engagement.

Future Work:

1. In the production environment, client load is received from multiple machines and the amount of load can vary with the client. Our current load testing uses a single machine to flood DL-Lambda. We can deploy Locust on a distributed environment and perform load testing. Some possible options are to deploy Locust on another Kubernetes cluster or a set of virtual machines or even as AWS lambda functions (serverless. As Locust only needs to run for a short amount of time).

2. We currently only support single query workloads per user request. However, users might want to perform multiple queries over a single image. This can also save some network bandwidth and resource utilization.
3. Frameworks such as DL-Lambda are susceptible to flooding-based attacks and DDoS. DL-Lambda currently doesn’t have security firewalls, and rate limiters to prevent such attacks.