Establishing a Release Engineering Pipeline for a Restaurant Sentiment Analysis System: A Structured Approach

Bryan He, Nektarios Evangelou, Zihau Xu, Marc Otten

May 2023

1 Introduction

In the current data-driven age, extracting valuable insights from data has become an essential strategy for enterprises. One such critical aspect of this strategy is sentiment analysis, an application of machine learning that classifies and interprets emotional content in textual data sources. This essay presents a systematic approach to architecting a release engineering pipeline for a restaurant sentiment analysis system, aiming to categorize customer reviews' affective tone.

The proposed framework involves a series of interconnected components, each tasked with distinct functions contributing to the holistic operational scheme. These components will be organized under an open-source entity, remla23-team12, comprising five repositories: model-training, model-service, lib, app, and operation. Ensuring public access to these repositories promotes transparency and accountability and facilitates constructive feedback and augmentation from the community. The following paragraph will outline the specific role and functionality of each repository. The model-training repository contains the machine learning training pipelines, while the model-service repository will contain the trained model's service wrapper. The lib repository will encapsulate a version-aware library to optimize resource utilization and promote reusability. The app repository will function as the user-facing component of the system, orchestrating all other elements into an integrated, interactive solution. Lastly, the operation repository will maintain all deployment-related files required for the Docker Compose & Kubernetes setup.

This design will incorporate automated workflows, containerization techniques, versioning mechanisms, RESTful APIs, and configurable parameters to ensure the seamless and efficient operation of the system. These considerations are not only intended to ensure smooth operation but are also devised to support the system's robustness, scalability, and maintainability. Further segments of this essay will describe each repository in detail regarding their functionality, responsibilities, and how they collaborate to create a comprehensive sentiment analysis solution for restaurant reviews. The main goal of this essay is to provide information on establishing a robust release engineering pipeline for a practical, industry-oriented application.

2 Docker compose migration to Kubernetes

2.1 Kubernetes and Prometheus

We migrated our Docker compose setup to a Kubernetes cluster. Kubernetes is an open-source platform for automating containerized applications' deployment, scaling, and management. The migration was done because Kubernetes is more agile and scalable in incorporating multiple cloud environments and clusters than Docker.

We migrated our Docker compose to Kubernetes using a 'deployment.yml' file, where we defined two Kubernetes deployments, 'flask-container-1-depl' and 'flask-container-2-depl', each corresponding to a distinct Flask container. A Flask container refers to a containerized application built using Flask, a web framework for building Python-based web applications. These containers are lightweight, isolated environments that package applications and their dependencies, allowing them to run consistently across different computing environments. The deployments manage the replicas of our application, ensuring that if a Pod goes down, another is created to maintain the service. Both these containers are automatically released by utilizing GitHub Workflow actions. These actions are a feature of GitHub that enables the creation of custom software development lifecycles (SDLC) workflows directly in the GitHub repository. Individual tasks, "actions," can be combined to create a custom workflow.

Furthermore, we implemented Kubernetes services for the corresponding deployments, 'flask-container-1-service' and 'flask-container-2-service' are responsible for distributing network traffic to the correct Pods. These services abstract the underlying Pods, creating a decoupled system where changes do not affect how the services will operate. The 'flask-container-1-service' is exposed outside our Kubernetes cluster using the type 'NodePort,' making our application accessible from outside the cluster. An 'Ingress' was also defined to manage external access to the services in our cluster. The 'flask-app-ingress' Ingress directs incoming traffic to the correct services based on the request's HTTP path. One path leads to our application, while another directs to our Prometheus instance, which facilitates access to monitoring data.

Prometheus is utilized to observe our applications' performance in real time. This functionality is integrated into our Kubernetes setup through a 'ServiceMonitor' object. The 'ServiceMonitor' selects the 'flask-container-1-service' to scrape metrics from, with an interval of one second. This frequent scraping ensures that our metrics are updated in near real-time, providing an accurate and current representation of our system's status. It contributes significantly to understanding user interactions and gauging the performance of the sentiment analysis model. Along with monitoring, Prometheus can create alerts when scenarios occur of unusually high service requests.

2.2 Helm and Grafana

Additionally, we provided the option to deploy our clusters using Helm. Helm is a Kubernetes package manager that helps define, install, and upgrade complex Kubernetes applications. It manages Kubernetes charts, which are packages of pre-configured Kubernetes resources. We have utilized Helm for our deployment, making installing the entire setup more than once in the same cluster possible. Helm can integrate Prometheus and Grafana.

Grafana is an open-source analytics and interactive visualization web application. It provided a dashboard for our project to represent our custom metrics graphically, giving us a more intuitive understanding of our web app's behavior. We created an advanced dashboard employing multiple visualization styles for Gauges, Counters, and more, leveraging functions like rate or avg for more sophisticated plots. We included the dashboard configuration in a ConfigMap to make them automatically available upon installing our web app. This approach circumvented the need for manual dashboard import, contributing to a smoother operational experience. Currently, our Grafana dashboard displays different kinds of information, such as the total prediction of our model over time, queries for the total prediction and the correct prediction, predictions rate over time, memory usage over time, and the CPU rate over time.

In both Kubernetes and Helm options, sensitive information, such as credentials required to pull our application's container images from the GitHub Container Registry, was handled securely. Instead of hard-coding these credentials into our files, they were references using 'imagePullSecrets,' which stores them separately in Kubernetes secrets. On the other hand, we used variables to secure our SMTP passwords to make it possible for alerts to be sent via email.

3 Expanding our services using Istio

3.1 Istio

Istio is an open-source service mesh that allows connecting, managing, and securing microservices. Istio provides a platform to manage services in a distributed manner, creating a network of deployed services, such as load balancing and monitoring, without requiring changes to the service code. The mesh is essentially an infrastructure layer for controlling service communication.

We have implemented traffic management in our pipeline to allow us to control the routing of the requests, enabling techniques like canary releases or A/B testing. Istio manages our deployments and traffic routing using Istio's Gateway and Virtual Service. We made the application accessible through IngressGateway. The setup allowed us to manage the routing requests based on the rules defined in the Virtual Service.

We again implemented Istio by incorporating it in Helm and as a standalone 'deployment.yml.' We have defined multiple versions of the flask-container deployments, such as 'v1', 'v2', and 'experimental.' Each version has different images pulled from the repository using 'imagePullSecrets.'

Furthermore, we have defined each deployment's services that expose the deployments to the Kubernetes cluster. We used Istio's 'DestinationRule' to divide the traffic among the different versions of the flask-container deployments. This setup enabled us to implement the canary release. The 'VirtualService' configuration defines how the requests are routed to different subsets or versions of the services. We use a combination of weight and match routing techniques to redirect traffic. The Gateway configuration handles incoming HTTP connections and directs them to the 'VirtualService,' which routes the traffic to appropriate services.

3.2 Shadow Launching & Rate Limiting

We have implemented the same logic for our Istio integration with Helm as our 'deployment.yml' for Kubernetes. However, since Istio has many other features beyond request routing, we also implemented a few different use cases in the form of Shadow Launching and Rate Limiting. Shadow launching is a technique where a new version of a service is deployed alongside the existing version, and copies of incoming requests are sent to both versions. However, only the response from the current production version is returned to the user. This allows us to observe the behavior of the new service version under real-world load conditions without affecting user experience. We accomplished shadow launching by using Istio's traffic mirroring feature. The configuration in the 'VirtualService' sends all requests to 'flask-container-1-service' while mirroring the same to the 'flask-service-2-service.'

On the other hand, rate limiting is a technique for controlling network traffic where the number of requests a client can make to a server within a specific time is limited. It is often used to prevent abuse, limit resource utilization, and maintain quality of service. We have configured Helm to implement rate limiting using an 'EnvoyFilter.' The Envoy proxy, part of the Istio service mesh, is configured to limit the rate of inbound HTTP requests to each Flask container. Our configuration allows ten tokens (requests) per 60 seconds, limiting the number of requests a client can send within that timeframe.

To ensure the shadow launch and rate limiting were working, we utilized some Istio add-ons for monitoring and visualization. We have used Prometheus for general monitoring of our backend, Jaeger, a tracing tool that monitors the requests in the data plane of our Istio installation, and Kiali, a visualization tool that allows exploring the cluster setup and inspecting our internal communication. Initially, we had some strange issues with injecting Istio's proxy into our Pods. Our Pods could not install Sidecar, which is important in order to inspect the traffic going in and out of our service mesh. Sidecar is a utility container in a pod that is bundled with the main application container. This was resolved by .... TODO

With the help of Kiali, we could inspect the traffic our Shadow Launch created and could see that the requests were also going towards 'v2' instead of only 'v1.'

FIGURE!

Furthermore, we experimented by sending the following curl command at least ten times to evaluate our rate-limiting implementation. Once we sent the eleventh request, we got a 429 error request for too many requests, which showed us that our implementation was working correctly.

CURL COMMAND

3.3 Documentation of the experiment

The webapp has a service that analyzes the sentiment of the review that a user provides and a value of 1 or 0 is returned. To get such a value, the user input is preprocessed before it is fed into a model. Currently, the textual user input is transformed into a Bag of Words (BoW) before it is fed into a Gaussian Naive Bayes model. When trying this model out, we have often seen that it returns the value 0 for input which carry positive sentiments. As a response to this, another model was trained using a different machine learning algorithm, namely Linear Support Vector Machine (Linear SVC). Moreover, this alternative model is not trained on BoW input, but term frequency - inverse document frequency input (tf-idf).

To decide whether the alternative model can replace the current model, we first have to gather analyze data about it's correctness. The metric gathered is the accuracy and count of the number of correct analyses. It is scraped by Prometheus when users click on the correct/incorrect buttons in the front-end applications.

Because the alternative model is experimental, it was decided to not expose a potentially less correct to all users at once. The canary release strategy is implemented by using Istio's Custom Resource definitions to set up a service mesh infrastructure. In total two services are deployed. The current model accessible to 85% of users and the remainder 15% are exposed to the experimental model.

Our falsifiable hypothesis is as follows: The accuracy of the experimental Linear SVC model is greater than that of the Gaussian Naive Bayes model, where the accuracy is the percentage of correct predictions out of the total number of predictions.

todos: - and elaborate the decision process. - Screenshots of a (Grafana or Prometheus?) dashboard can be used to illustrate how the decision process is supported by data.

4 Organizing our model training pipeline using DVC