



University of Piraeus

Department of Digital Systems

Postgraduate Programme "Information Systems & Services"

Running Kafka clusters on Kubernetes

From a Proof-of-Concept to a Production-Grade Deployment

(Λειτουργία Kafka Cluster σε Kubernetes περιβάλλον)

Dimitris Ntosas

Author

Contents

1. Abstract	4
1. Abstract (GR)	5
2. Introduction	7
2.1 From Monoliths to Microservices	7
2.2 Kubernetes: Containerized Workload Orchestrator	17
2.3 Kafka: Distributed Pub/Sub System	24
2.4 The benefits of delivering Kafka in Kubernetes	27
3. Implementation	29
3.1 Identification of the problem we want to solve	29
3.2 Technologies we are going to use	31
3.3 High-Level Design	33
3.3.1 IaaS Layer	36
3.3.2 PaaS Layer	40
3.3.3 SaaS Layer	44
3.4 Specifications	47
3.4.1 Kubernetes Cluster	47
3.4.2 Kafka Namespace	51
3.4.3 Monitoring Namespace	54
4. Results	57
4.1 Unit Test	57
4.2 Load Test	60
4.3 Stress Test	64
5. Future Work	68
7. References	70

1. Abstract

Software design patterns have changed radically during the last decade. The emerge of Cloud Computing capabilities accompanied by more demanding business requirements, made the industry moves towards new architectures that promised to bring efficiency to the next level.

As a consequence, functionalities like Message Queuing and Containerized Workload Orchestration became common requirements in modern software engineering. Kafka as Message Broker system and Kubernetes as Container Orchestrator bring great features to serve the above needs and they are already tested on mission-critical environments.

Problems begin when it comes to maintain and use these systems. Kafka and Kubernetes along with their powerful offerings, introduce great complexity for both Administrators and End-Users that usually act as a burden for their adoption in existing or new setups.

We want to challenge these problems by creating a Platform that combines Kafka with Kubernetes and provides their functionalities as a Service. An abstraction interface that brings simplicity and confidence in the way applications and users interact with these technologies.

Our goal for Applications utilizing this platform is to rely on benefits starting from Reliability and end-to-end Performance to enhanced Fault Tolerance.

Our aim for users acting as Operators is to reduce the burden of manual operations and bring a smoother experience to the administrative process. For users acting as Software Engineers, we want to provide the capability of an accountable reduction of complexity to their Deployment Patterns and a minimalistic way of declaring Message Queuing dependencies. Once configured, Kubernetes, Kafka, and other applications deployed through our Platform will scale and self-heal without any manual intervention.

1. Abstract (GR)

Οι προσεγγίσεις στο σχεδιασμό λογισμικού άλλαξαν ριζικά την τελευταία δεκαετία. Η εμφάνιση των δυνατοτήτων του Cloud Computing, συνοδευόμενη από πιο απαιτητικές επιχειρησιακές απαιτήσεις, έκανε τη βιομηχανία να κινηθεί προς νέες αρχιτεκτονικές που υποσχέθηκαν να φέρουν την απόδοση του λογισμικού στο επόμενο επίπεδο.

Κατά συνέπεια, λειτουργίες όπως το Message Queuing και το Containerized Workload Orchestration, έγιναν κοινές απαιτήσεις στη σύγχρονη μηχανική λογισμικού. Το Kafka ως σύστημα Message Broker και το Kubernetes ως Container Orchestrator φέρνουν εξαιρετικά χαρακτηριστικά για να εξυπηρετούν τις παραπάνω ανάγκες και έχουν ήδη δοκιμαστεί σε κρίσιμα επιχειρησιακά περιβάλλοντα.

Τα προβλήματα αρχίζουν να εμφανίζονται στη συντήρηση και τη χρήση αυτών των συστημάτων. Τα Kafka και Kubernetes, μαζί με τις ισχυρές λειτουργικότητές τους, εισάγουν μεγάλη πολυπλοκότητα τόσο για τους διαχειριστές όσο και για τους τελικούς χρήστες που συνήθως λειτουργούν ως ανάχωμα για την υιοθέτησή τους σε υφιστάμενες ή νέες εγκαταστάσεις.

Θέλουμε να δώσουμε λύση αυτά τα προβλήματα δημιουργώντας μια πλατφόρμα που συνδυάζει τα Kafka και Kubernetes και παρέχει τις λειτουργίες τους ως Υπηρεσία. Μια διεπαφή αφαίρεσης που προσφέρει απλότητα και την αυτοπεποίθηση στον τρόπο με τον οποίο οι εφαρμογές και οι χρήστες αλληλεπιδρούν με αυτές τις τεχνολογίες.

Ο στόχος μας για τις εφαρμογές που χρησιμοποιούν αυτή την πλατφόρμα είναι να εκμεταλλεύονται οφέλη που ξεκινούν από την αξιοπιστία και την απόδοση έως τη βελτιωμένη ανοχή σφαλμάτων.

Ο στόχος μας για χρήστες που ενεργούν ως Διαχειριστές είναι να μειώσουμε το βάρος των χειρωνακτικών λειτουργιών και να επιτύχουμε μια ομαλότερη εμπειρία στη διοικητική διαδικασία. Για τους χρήστες που ενεργούν ως Μηχανικοί Λογισμικού, θέλουμε να παρέχουμε τη δυνατότητα μιας σημαντικής μείωσης της πολυπλοκότητας στα πρότυπα ανάπτυξης τους και έναν μινιμαλιστικό τρόπο δήλωσης των εξαρτήσεων των εφαρμογών τους.

2. Introduction

2.1 From Monoliths to Microservices

During the last two decades, software was designed and deployed as monoliths. In monolithic applications, single instances performed all the business logic functions.

Most of this software were Enterprise applications, which in most cases were utilized by a single company or organization. The development, deployment, and maintenance was an internal responsibility, and its lying infrastructure usually was on-premise. Common examples of such applications were in-house billing or accounting. The constant definitions or parameters such as business scope, size, and security constraints could ensure high process adoption and performance. High availability or downtime-less systems were not a firm requirement back then [1, 2]

But, technological breakthroughs during the last two decades extended the limits of what software applications are capable of offering. Performance on complex tasks like Business Intelligence, Customer Management, Traffic Analytics was boosted significantly, leveraging both better hardware and more efficient software patterns and languages.

Operators and developers needed to find new ways of efficiently managing this complexity as the codebase's size grew. At the same time, we got an exponential bump in Internet usage. That was a catalyst and a newborn dimension that eventually produced a new category of software and pattern, which is widely known as web applications.

The fact that these pieces of software could be accessible from everywhere in the world, from every device, with just using a browser, arose a new requirement for development lifecycle: the need of having 0 downtimes. So, what we today call Resilience, started to be on an

engineers' everyday life that built systems meant to be served on the web, with unexpected fluctuations of workload or network.

Monoliths had straight limitations that could not follow the new development requirements that arose. The shared ownership of the codebase made engineering teams grow big enough to introduce several headaches in the process. To tackle this, teams started to split the monoliths into smaller parts to handle one business function each time and finally get closer to a Service Oriented Architecture [2, 6].

Going Cloud Native Era

At the same time that the world started to migrate to more fine-grained architectures like SOA and EDA [50, 51]. A new hype was born that was promising simplicity and efficiency on leveraging compute resources: the Cloud!

The ever-increasing requirements of Web Application Development for speed, flexible scaling, and around-the-clock availability created a challenge in providing the required on-premise infrastructure in an economically feasible way. This challenge led us to the rise of on-demand cloud computing platforms.

Cloud computing platforms such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud have created a global network of data centers based on an economy of scale approach. Coupled with central management and high-level engineering, the end-user can practically lease highly available and reliable compute resources for as long as required and at any level of scale [2, 4].

The two driving factors behind Cloud Computing adoption have been: The Pay-per-use model, which frees companies from the initial capital cost of building infrastructure as well as the abstraction of the physical

infrastructure to compute resources which answers the challenge of keeping up with Moore's Law.

Open Issues

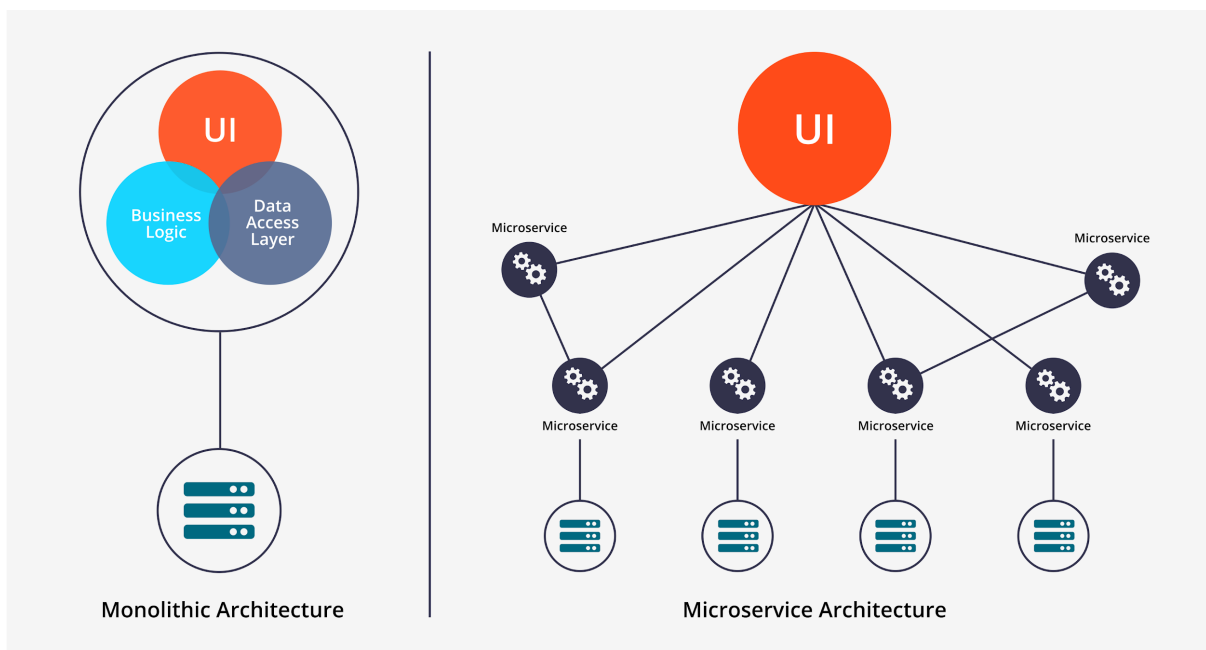
Handling the physical infrastructure through Cloud technologies is only one of the problem's facets. There are numerous issues in an application's development life-cycle that need to be considered, such as development and deployment[6].

- **Increasingly large build times:** Monolithic applications increase their codebase over time, resulting in ever-growing build times, this hinders development speed and in some cases can also affect reliability since quick changes (eg a rollback after a failed deployment) in the software are not possible.
- **Development Speed:** A monolithic application encourages tight coupling, thus even small changes to the code base can cause radical and unwanted changes in the application, in an enterprise context, a bug introduced by one team can negatively impact every other team. This environment creates a culture of long non-incremental deployment cycles contrary to what Agile Software development proposes (evolutionary development, early delivery, and continual improvement)
- **Inflexibility to Failure:** By construction, a monolithic application will become unavailable if any of its functions fail, however, this behavior can become extremely disruptive while the codebase grows. An optimal solution should allow us to define precise error boundaries (that allow propagation of errors only inside the relevant subsystems) and allow our application to function at the best of its capacity while some subsystems might be failing.

- **Resource Overprovisioning:** In the event of suddenly increased workload for a particular business function, our only option to handle the load through a monolithic application is to multiply the full instances and load balance between them. As a consequence, we get a non-optimized utilization of compute resources and in the context of Cloud Computing suboptimal consumption leads to increased costs. One would prefer a scenario where we can flexibly scale only the parts of the application that are under great load.

Microservices

Industry realized that a coordinated engineering contribution should take place in order to avoid the problems with the monolithic architectural style discussed above, resulted in the development of a new architectural style: Microservices [1, 2, 6].



High-Level Architecture style comparison

Following this style, the monolith is decomposed into an ecosystem of loosely coupled services (smaller applications).

Every and each for these new modules of software is now responsible for a specific well-defined problem area. When implementing business

requirements we do so by creating services that can communicate and aggregate data between them.

Decomposing software into smaller modules makes it easier to build, understand, and test. Problems mentioned above for monoliths are addressed by design.

Comparing to the Monolithic approach, a microservices strategy has the following advantages:

- a) Well-defined microservices have a considerably smaller and less complex problem space (compared to an all-encompassing monolith) thus making it easier to argue, design and implement them.
- b) When a change needs to be introduced only the affected microservice needs to be rebuilt. Reduced deployment time is one of the main goals when trying to achieve continuous delivery and deployment.
- c) On a well-designed system failure of a separate microservice should only cause downtime on that service and not to the entirety of the system. However, this introduces development cost since each service should be designed to work in a network under the requirement that other services might become unavailable.
- d) Fluctuation in Compute requirements can be optimally managed: When a system is decomposed to microservices we can scale only the affected services and not the entirety of the system.

Challenges that Microservices introduced

Microservices pattern was like a blast that solved by-default and by-design several problems that Monoliths had but they also introduce fresh new challenges:

- **How to achieve optimum utilization of Cloud Resources?** As also mentioned before, in Microservices, workloads are dynamically scaled to the required number of replicas so it's essential to think of a way of optimizing the utilization of cloud resources. In simple words, fine-grained control over how cloud resources are being utilized. Microservices architecture is certain to have too many moving parts. Manual control is not applicable when it comes to the usage of all the cloud resources. Designing and implementing automation that takes care of these controls, becomes a new challenge that we can't avoid but tackling it.
- **How do we manage dozens of Deployment Operations?** Going Microservices, a single application is split into multiple smaller parts. Attention must be taken to ensure that communication between the workloads is still reliable and efficient. At the same time, the operating system layer of the host machines must comply with the newly introduced required dependencies.
- **Complexity can now follow exponentially the growth of services number,** and we can assume that for the majority of early adopters, regular operations -as we knew them- can easily become a nightmare. A robust approach with a solution-suite that enables operators to efficiently handle the complexity of the introduced operation has become indispensable.
- **How do we guarantee a reliable Communication Bus?** The Microservices approach often brings a large number of services that need to communicate and at the same time to be loosely coupled and autonomous. As a result, software development tends

to adopt the Observer pattern, and the need for robust publish-subscribe messaging systems arises.

The solutions to these new challenges mainly required two key innovative technologies.

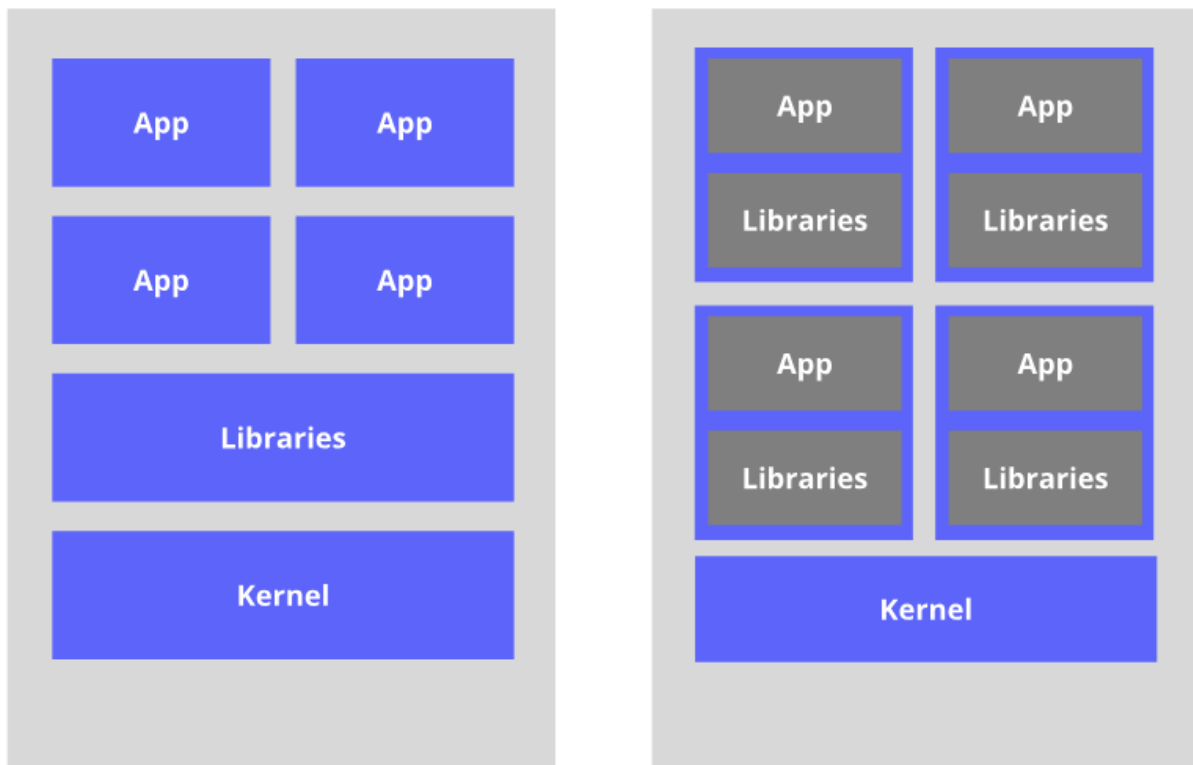
Containers to enable the efficient packaging and deployment of distributed applications and a Container Orchestrator System that could handle the management of multiple Containers relationships.

Containers

The container is virtualization at the OS layer where the kernel enables and manages the presence of multiple instances at user-space [7, 8].

In simple words, imagine a container as a fully functional virtual machine, running inside another host machine (host). Every jail like this (container) is now fully isolated from another jail lying on the host, and of course from the host.

Containers have their own namespaces and they have isolation from the file system to process and compute resource usage. Docker, Containerd, and rkt are some of the most in-use container runtime daemons.



Monolithic/Legacy vs Containerized deployment of software

Containerization mainly introduces two main features:

- **Application and Infrastructure decoupling.** Life without containers has as a common practice to deploy software applications directly on host machines. That means operators had to include the logic of operating applications inside machine configuration, libraries, system services, and lifecycle. A trivial puzzle between application and host which eventually concluded on the host machine to be transformed into an application itself. With Containerization in place, applications are delivered as packages into the host machine to run. These “packages” include by default all dependencies into self-isolated containers that the host machine just needs to run. The application and the host OS are now completely decoupled, enabling users to deploy the application to any host without worrying about configuring underlying constraints. [9].

- **Resources Isolation.** Decoupling applications from underlying hosts removed a lot of friction on deployment operations. In addition, another major advantage of Containerization is the capability to isolate compute resources between different containers. Think about a faulty container that returns errors. Now, it is totally isolated and can not affect other containers running on the same host. In addition, each container can be guaranteed a certain amount of CPU and Memory [9].

The industry has already understood the value this method of packaging and deployment brings, and we can say it is now a widely adopted pattern.

Container Orchestration System

As the industry highly adopted the usage of containers the need for a system that could cope with the management of multiple container relationships arose. The name of this piece of technology was Container Orchestration System.

On a high level, container orchestrators should have all the needed capabilities of handling containers' lifecycle. Talking about the more specific scope and steps on how this layer would fit on the whole platform, how to split responsibility, etc, it's totally up to the team that owns and designs such a system. The good thing is that the flexibility and extensibility that such a system offers, allow operators to build based on various approaches [10].

In simple words, such a system won't be only present to control the start or stoping of a container, but should be able to provide more abstracted functions to address the following outliers [10, 11]:

- Applications must be highly available and deployed in a manner that eliminates downtime risk, regardless of the complexity of their functions.

- Applications must be resilient to sudden load spikes, regardless of the amount of fluctuation introduced.
- The system must be able to optimally use the cloud compute resources available.
- Deployment operations like rolling upgrades or rollbacks should take place in a smooth manner.

The majority of engineering teams in the industry have faced the above problems, implementing different solutions or combinations of them. That gave the community the lead to think of a project that can be utilized to solve the above.

As requirements were almost the same for everyone, designing an orchestrator that addresses all of them together seemed like an efficient approach that could benefit the industry long-term.

Over the past decade, Google already started to work on many projects that wanted to solve deployment problems. The outcome of this work was the so-called Kubernetes that they donated to the community. Kubernetes promised to offer a bundled solution for teams operating on containerized environments [17], backed by the whole experience that an organization in the scale of Google can provide [16].

2.2 Kubernetes: Containerized Workload Orchestrator

What is Kubernetes?

Quoting from official Documentation [12]. *Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.*

The name Kubernetes originates from Greek, meaning helmsman or pilot. K8s as an abbreviation results from counting the eight letters between the "K" and the "s". Google open-sourced the Kubernetes project in 2014. Kubernetes combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community.

It is designed to effectively solve many of the issues we outlined earlier.

- i) Kubernetes can run containerized applications of any scale without any downtime.*
- ii) Kubernetes can self-heal containerized applications, making them resilient to unexpected failures.*
- iii) Kubernetes can auto-scale containerized applications as per the workload, and ensure optimal utilization of cloud resources.*
- iv) Kubernetes greatly simplifies the process of deployment operations.*



Kubernetes Logo

In summary, Kubernetes is a system capable of performing reliably complex deployment/management operations with a bunch of commands or a few lines of code.

As a project, was originated by Google and now is donated and maintained by Cloud Native Computing Foundation (CNCF) which is a foundation with core scope to ensure that cloud-native patterns would be universally accessible.

What problems is Kubernetes designed to solve?

Kubernetes can orchestrate and hold the desired state on compute, network, and storage given a set of constraints from a user. It eliminates the dependency of manual intervention and automates the process of keeping an application highly available while utilizing allocated compute resources in the most optimized way [12, 13].

- **Dynamic Service Discovery.** Your environments may consist of dozens of workloads, while you add or remove them on demand. Kubernetes is taking care to provide this discovery natively and reliably for the whole cluster.
- **Load Balancing.** Each of the workloads gets a unique LB targeting all backend pods, providing a simple interface to distribute traffic among all targets, enabling the virtually infinite horizontal scaling.
- **Autoscaling.** Handle the compute load surge of workloads in a horizontal manner. Scaling out replicas when traffic is increased while gracefully scaling when the surge is finished to avoid unessential costs. Kubernetes handles all these operations automatically for users.
- **Self Healing.** Kubernetes watches all backend endpoints for knowing its state through multiple ways and if a service goes broken, it automatically applies recovery actions.
- **Smooth Rollout and Rollback.** Kubernetes can deploy any kind of application as a rolling operation either by one-by-one or by replacing multiple instances at a time with just a command or a bunch of configuration lines of code. At the same time, these operations also respect all the other features like self-healing during the rolling.
- **Environment, Configuration and Secret Management.** Kubernetes implements two kinds of resources that are meant to store both sensitive or non-sensitive data. ConfigMap is a flexible volume that can be read in multiple ways, ideal for storing configuration data. Secret is a protected volume bounded with core k8s security controls that is ideal for storing encrypted data like passwords etc

where we want to minimize the risk of exposing them by mistake. At the same time, a Pod is capable of holding a different environment per Container so users can inject their variables of choice and provide an easy interface to comply with 12FA app requirements.

- **Storage Management.** Kubernetes implements an interface (CSI) for effectively managing the state of workloads. It provides richful attribution and isolation between storage claim, attaches or allocation. Firstly, this was well integrated only with famous public Cloud providers but now Kubernetes extended this compatibility with almost all kinds of known storage solutions. It also allows operators to define storage constraints per scope via StorageClasses.

Kubernetes Terminology

Kubernetes defines a large number of abstract objects. Here, we will only discuss those Kubernetes objects that are essential for understanding our implementation [14]. Quoting from source [15]:

- **Pod:** *We know that through Kubernetes, we could run containerized applications. Instead of abstracting a single container as a Kubernetes object, Kubernetes defines pod, which is a group of one or more containers. There is an advantage arising by making this choice. For simpler cases, each pod in the system could represent a single container. But, whenever there is a need to deploy additional capabilities that are not directly related to the core business functionality of the container – like support for logging, caching, etc – we have an option to package these additional capabilities into separate containers and place them in a single pod. This ensures they always stay logically together. Pods are the smallest deployable units of computing that can be created and managed in Kubernetes. It is the place where the actual application code implemented by the end-user runs. Each pod has its own IP address and is completely decoupled from the host. [15]*

- **Service:** *In Kubernetes, pods are volatile. To ensure high availability and optimum use of computing resources, Kubernetes could dynamically kill and create pods. Because of this, the IP address of a pod is not a reliable way to access the business functionality offered by the pod. Instead, Kubernetes recommends using a service to access the business functionality. Kubernetes service is an abstraction that defines a logical set of pods and a policy to access them. Every Kubernetes service has an IP address, but unlike the IP address of a pod, it is stable. A Kubernetes service continuously keeps track of all the pods in the system and identifies the pods it is expected to target. Whenever a request to access a particular business functionality reaches the service, it will redirect the request to the IP address of one of the pods that are active in the system at that point in time. Ideally, to access the pods from outside the cluster, one must use Ingress. As of now, however, the Kubernetes Ingress feature is still in beta. Thus, in this example, we will use a service to expose the traffic externally as well. [15]*
- **PersistentVolume and PersistentVolumeClaim:** *Managing storage is a distinct problem from managing to compute power. Kubernetes defines two key abstractions to handle this problem, i.e. persistent volume and persistent volume Claim. In Kubernetes, a persistent volume is a piece of storage in the cluster that has been provisioned to be used by the cluster for its storage requirements. A persistent-volume claim is a request by an application to consume the abstract storage resources declared through persistent volume. To make persistent storage available to the applications running inside Kubernetes, one should first declare persistent volume and then configure the application to make a claim to use that volume. [15]*
- **ConfigMap:** *Configmap is a Kubernetes abstraction meant to decouple environment-dependent and application-configuration*

data from containerized applications, allowing them to remain portable across environments. [15]

- **Secrets:** *A secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Putting such sensitive information in a secret allows for more control over how it is used and reduces the risk of accidental exposure. [15]*
- **Deployment:** *Deployment is an abstraction meant to represent the desired state of an actual deployment on Kubernetes. A deployment object typically contains all the information required – the location to obtain and build containerized applications, the configuration of pods expected to package and run these containers, the number of replicas of each pod that should be maintained, the location of application configuration in terms of config-maps and secrets meant to be used by the containers, the configuration of data storage (if the application needs persistent data storage). All of these could be declared inside deployment. Although it is possible to create individual pods and services in Kubernetes, it is recommended that one uses deployment to manage deployments. By using the deployment object, typical operations like roll-out, roll-back, and monitoring are greatly simplified. [15]*
- **CRD:** *In the Kubernetes API, a resource is an endpoint that stores a collection of API objects of a certain kind. For example, the built-in pods' resource contains a collection of Pod objects. The standard Kubernetes distribution ships with many inbuilt API objects/resources. CRD comes into the picture when we want to introduce our own object into the Kubernetes cluster to full fill our requirements. Once we create a CRD in Kubernetes we can use it like any other native Kubernetes object thus leveraging all the features of Kubernetes like its CLI, security, API services, RBAC, etc. The custom resource created is also stored in the etcd cluster with proper replication and lifecycle management. CRD allows us to use all the functionalities provided by a Kubernetes cluster for our*

custom objects and saves us the overhead of implementing them on our own. [58]

What Kubernetes is not designed to do

Kubernetes is a fully-featured suite for managing containers. However, there are many things that operators are tempted to believe are part of its core functionality but the truth is that the following are expected to be integrated with tools/processes outside of the orchestrator itself [18]:

x) **Kubernetes does not provide a network.** It just has an interface (CNI) that configures an already implemented network layer so as to provide transparent communication between Pods and Nodes without NAT.

x) **Kubernetes is not a CI/CD system.** Building, testing, and delivering a codebase are not features of any orchestration system. What Kubernetes does well in those cases is to provide a simple interface for automation systems like Jenkins to easily integrate with.

x) **Kubernetes does not provide application dependencies.** Databases, caches, event buses, or storage systems are not built-in components of k8s. These dependencies can use Kubernetes as runtime but they need to be deployed separately from the orchestrator itself.

x) **Kubernetes does not provide monitoring constraints like Metrics, Alerting, or Logging.** There are built-in mechanisms that help users easily ingest logs and such kinds of information but these are only proof-of-concept functionalities and their purpose is to just provide an interface for external tools to consume and handle. Prometheus and Fluentd are some popular projects that aim to contribute to such functions.

Kubernetes brought great value to its end-users, giving them a robust platform with interoperability with 3rd tooling by default and by design so

it's important to utilize its innovation instead of exploiting it as a one-for-everything solution.

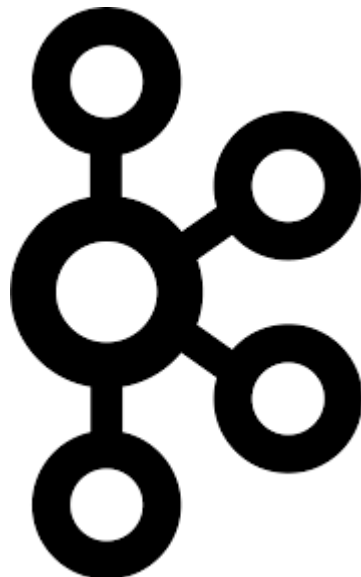
2.3 Kafka: Distributed Pub/Sub System

What is Kafka?

Apache Kafka is a message bus system that allows high throughput of messages to be streamed and be published or subscribed on-demand. Its distributed nature enables users to process efficiently and reliably huge amounts of data volumes. [19]

Kafka guarantees the prevention of data loss by persisting its data on the disk and also replicating it to different physical or logical locations. It is built with a dependency on ZooKeeper which is a known stable and performant service for state sync.

The core principles that the project wants to comply with, are to be a scalable low latency platform for processing in real-time big amounts of data streams, aiming to the needs of both simple implementations and demanding enterprise ones [20].



Kafka Logo

Kafka Features

Kafka as a messaging platform in addition to other platforms also provides [19, 21]:

Scalability: Kafka is able to reliably and performantly scale in multiple dimensions: On the producer/consumer side, on event processing, and event connectors. In other words, Kafka scales easily without downtime even with huge volumes of data.

Fault Tolerance: Kafka is able to efficiently handle failures both on the control and data plane.

Reliability: Kafka offers Durability and Replication as it uses Distributed commit logs, which means messages persist on disk as fast as possible across several nodes. This kind of distribution and partitioning guarantee its reliability constraints.

Performance: Kafka can support high throughputs both for publishing or subscribing to message streams. Even for many Tbps of traffic, Kafka implementation and resource utilization allow it to maintain the same performance stability.

Extensibility: Kafka offers a very pluggable interface to develop additional features. At the same time, Kafka offers a simple and richful way to write your own connectors for it.

Data Transformations: Kafka can provision transformed data streams given a specific input from producers.

Kafka Terminology

Kafka defines a large number of abstract objects [21]. Here, we will only elaborate on those Kafka objects that are absolutely essential for understanding our implementation.

- **Broker:** In real life, a broker is an individual who takes care about transactions between two or more parties and takes a commission upon transaction execution, a facilitator between the two sides. In Kafka's terms, actual nodes called brokers as their prime responsibility is to receive messages from producers and let consumers fetch them.
- **Cluster:** A set of Brokers acting as a highly available distributed system.
- **Topic:** A log of streamed records. Every message that arrives in a Broker is written as a sequel in a topic.
- **Partition:** Kafka facilitates the data sharding technique to spread the load for a specific Topic. Partition is a subset of a given topic. [53]
- **Replica:** Kafka is able to produce Replicas per Partition spread on multiple Brokers to ensure fault tolerance on the cluster. Every Partition has a Leader that is responsible for taking write requests and read requests to keep consistency and if Leader fails, a Replica will take its role to continue the operations.
- **Producer:** Any external application that delivers messages to a specific Partition of Topic.
- **Consumer:** Any external application that fetches messages from a specific Partition of Topic.

2.4 The benefits of delivering Kafka in Kubernetes

Kafka is designed to offer by default High Availability, Fault Tolerance, and Horizontal Scalability. By deploying Kafka into Kubernetes we can take advantage of Kubernetes design principles to create a fine-grained Platform for modern software environments [22].

Let's break down principles that mainly describe the value of this Platform:

Portable: Kubernetes offers a universal way of creating a compute platform as it can be deployed anywhere (cloud, on-prem, etc). It provides a consistent runtime, from major public cloud providers to your personal computer. Software packages that are deployed on k8s have the same behavior and can be ported as-is to a different underlying environment without any certain restriction for the type of application. While the primary focus was to tackle the complexity that microservices brought to the game, even monoliths can be deployed on k8s and take advantage of all capabilities that the orchestrator offers. Kafka will be our candidate here.

Flexible: Kubernetes has no restrictions on how many moving and custom parts operators can combine to build their platform. In simple words, k8s offers the essential flexibility to build the features needed alongside existing infrastructure layers like monitoring, network, etc. Users can use Kubernetes for running their applications while using a 3rd party service for logging and the list goes on. Provisioning a functionality on a platform built on k8s is just a manifest away!

Extensible: Kubernetes exposes an interface for each feature available to use or extend. This allows users to implement new functionalities on top of existing ones and is also a reason that there are dozens of add-ons already developed for Kubernetes from the community.

Each of these principles adds great value to the end-user who wants an environment with Kafka to work on. Portability enables users to test or run applications on different environments, preventing vendor lock-ins. Flexibility and Extensibility enable the easy customization of functions that are not present by default but are essential for system scope.

In simple terms, developers **gain the freedom to choose the exact development tools and frameworks that are essential to comply with business requirements without the friction of putting effort into infrastructure and deployment.**

3. Implementation

3.1 Identification of the problem we want to solve

The problem here is that Kubernetes and Kafka combined, introduce great complexity in both the Administrators and Stakeholders. If we include in here, essential components like monitoring, etc, a whole expertise team may be a dependency to handle this end-to-end.

To deal with that, we want to create an abstraction interface that leverages Kafka as well as Kubernetes capabilities to provide them as a Service. This interface will feature Kafka resources as well as the application's dependencies, to be declared as Kubernetes Manifests. In addition, the end-to-end pipeline definitions as a code will introduce great complexity reduction.

Our aim is to reduce the burden of manual operations. Once configured, Kafka and other applications deployed through our Platform will scale and self-heal without any manual intervention. Kubernetes could be integrated with a Continuous Integration (CI) pipeline, allowing a code change committed by a developer to be deployed onto the test environment automatically.

This kind of automatability will ensure that manual work necessary for the maintenance of a large-scale application is kept at a minimum. This will allow a relatively small team to successfully maintain a large-scale, distributed application deployed on the cloud.

In order to better understand the solution proposed [22], we first need to understand which will be the system's (1) Offerings (2) Actors and (3) some of the Use Cases:

Offerings

1. **Event Bus Interface:** Kafka that provides an abstraction for the usability and configuration of itself.
2. **Observability Interface:** Prometheus and Elasticsearch that provide an abstraction for scraping and monitor deployed applications in the Platform.

Actors

1. **Applications** deployed in the Platform: Asks guarantees for the offerings.
2. **Platform Operators** as Owners: Responsible for reliability and configuration.
3. **Software Engineers** as End-Users: Stakeholders of the offerings.

Workflows / Use Cases

1. **Platform Operator** declares a Kafka Cluster as Kubernetes Manifest.
2. **Platform Operator** does rolling upgrade operations at Kafka Cluster through Kubernetes control plane.
3. **Platform Operator** does scaling operations at Kafka Cluster through Kubernetes control plane.
4. **Software Engineer** develops an application and asks for Kafka dependencies and Prometheus scraping for metrics exposure.
5. **Application** acts as Producer declare its Kafka dependencies as part of Helm Chart as Kubernetes Manifest.
6. **Application** acts as Consumer declare its Kafka dependencies as Deployment's environmental variables.

3.2 Technologies we are going to use

Infrastructure

- We are gonna leverage a **public Cloud Provider** (AWS) to obtain the **Compute Resources** needed [23].
- For **CPU and Memory**, we are gonna use pure **EC2** which is the cloud provider's definition for a virtual machine running on Xen Hypervisor.
- For **underlay Networking** (Routing Tables, Subnet Management, etc), we are gonna use plain **VPC** which is the cloud provider's definition for a virtualized namespace of network functions.
- **No managed** Platform or Software services will be used.
- Although it needs to be clear, the above-chosen Infrastructure specifications **are not hard dependencies** for the actual Implementation. We are free to switch between any system that supports **x86** or **ARM** architectures, from Cloud to Bare Metal instances. The final deployment will provide the same **functionality** with the **same configuration applied**, across any platform. [48, 49]

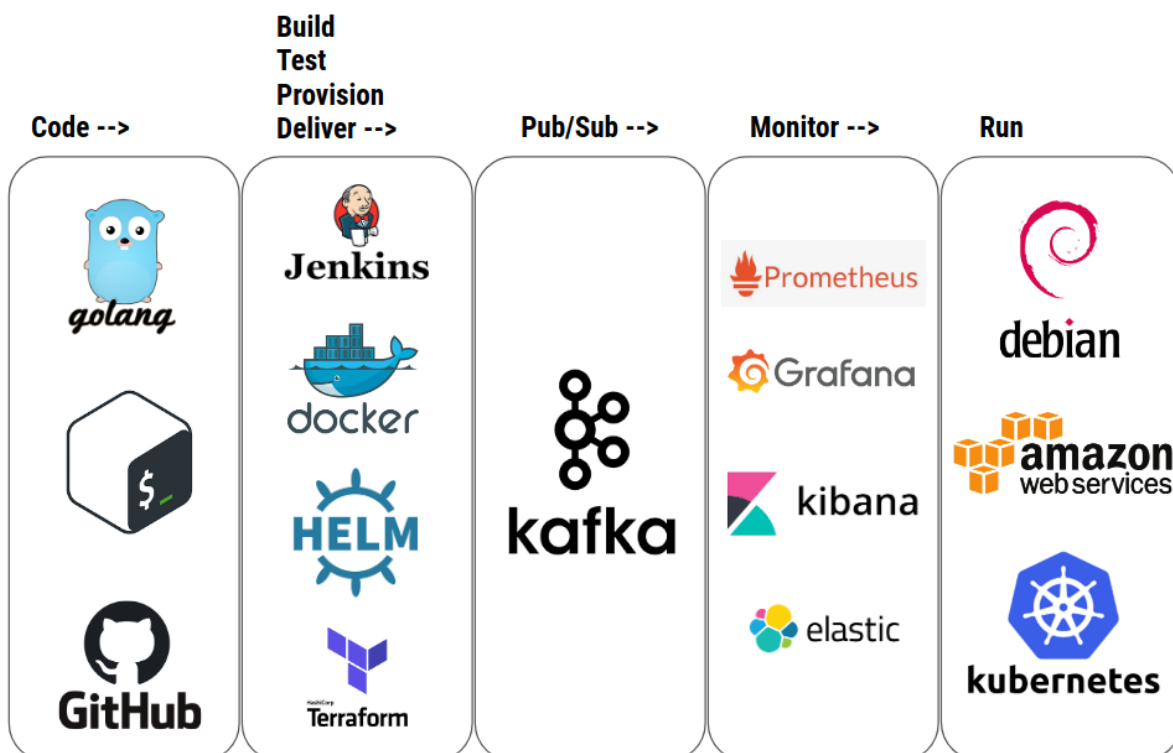
Platform

- You are going to use **Debian** as Operating System. Debian offers great flexibility in optimizing and configuring Linux Kernel variables to meet the needs of a heavy Network/CPU workload [24].
- **Kubernetes** as Container Orchestrator and **Kafka** as Event Bus which we have already thoroughly explained [12, 19].
- **Prometheus** to scrape and store Metrics which is a robust time-series database with an even more powerful query language [25].
- **Elasticsearch** to collect and store Logs which is a highly-available NoSQL database, nice fit to store our document structured data [26].

Tooling

- **Git** as Version Control [27]
- **Docker** as Container Daemon [28]
- **Terraform** as Infrastructure as a Code [29]
- **Kops** and **Helm** for Kubernetes Provisioning [30, 31]
- **Jenkins** as CI/CD [32]
- **Grafana** to visualize Metrics [33]
- **Kibana** to visualize Logs [34]
- Several benchmarking tools (**Vegeta** / **Netperf** etc)

The above technologies will be fed from their corresponding repositories, stable channels. Versions will be point pointed to the **latest stable** releases but **not** always to **LTS** ones. In addition, they all are **Open Source**, and their source code repo links can be found in the References section.



Technologies Overview

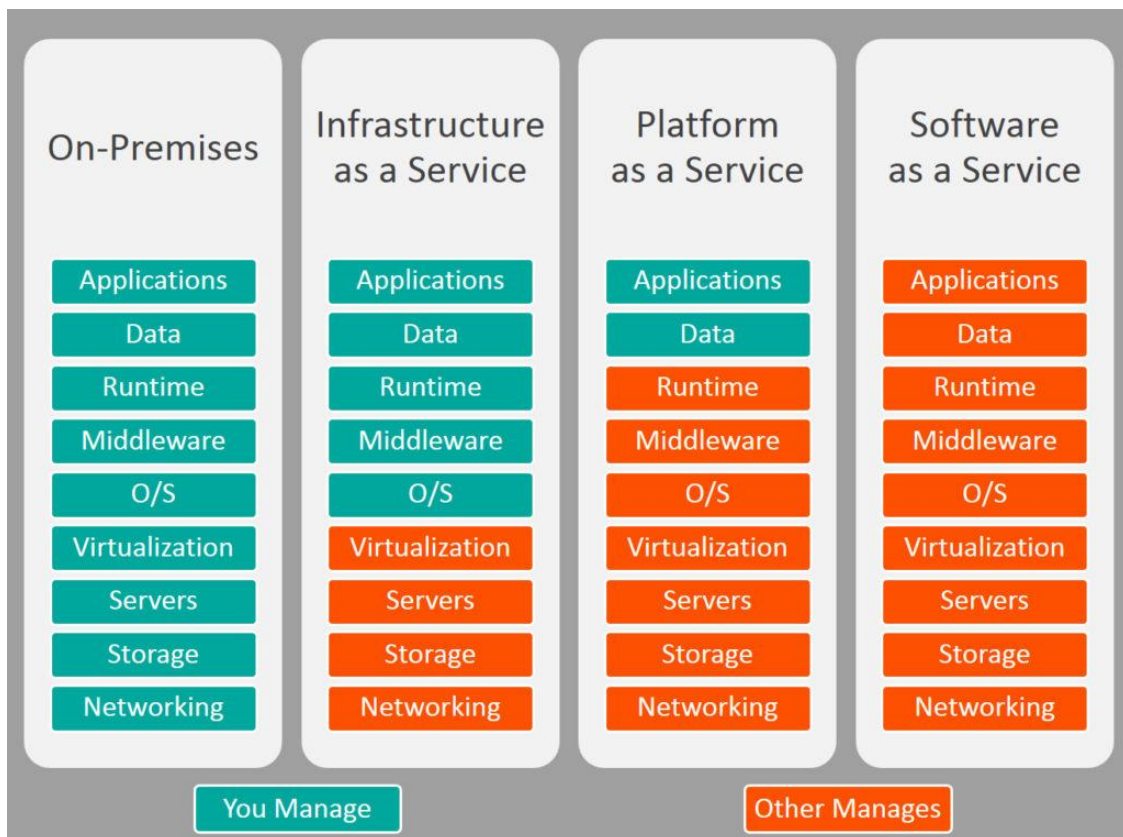
3.3 High-Level Design

In order to better describe and design our Implementation, we will slice it into three main layers:

- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)
- Software as a Service (SaaS)

These definitions were introduced during the last decade by the engineering community so as to have a more descriptive common language in the new environments the cloud computing era brings. [35]

When we refer to a resource and define in which of these layers the resource exists, we have already described (1) the entity which is responsible for managing this resource, (2) the possible dependencies from other layers, and (3) the offerings that would be provided.



On-Premise vs IaaS vs PaaS vs SaaS Comparison

In the above image, you can see a summarized view of how these layers are sliced into.

By adopting this model we can create ownership/administration isolation and declared guarantees of abstracted functionality. When a declared guarantee is violated, we can easily identify in which layer the violation takes place and address the issue without the need to track down all the components of such complex architectures as the modern ones [35].

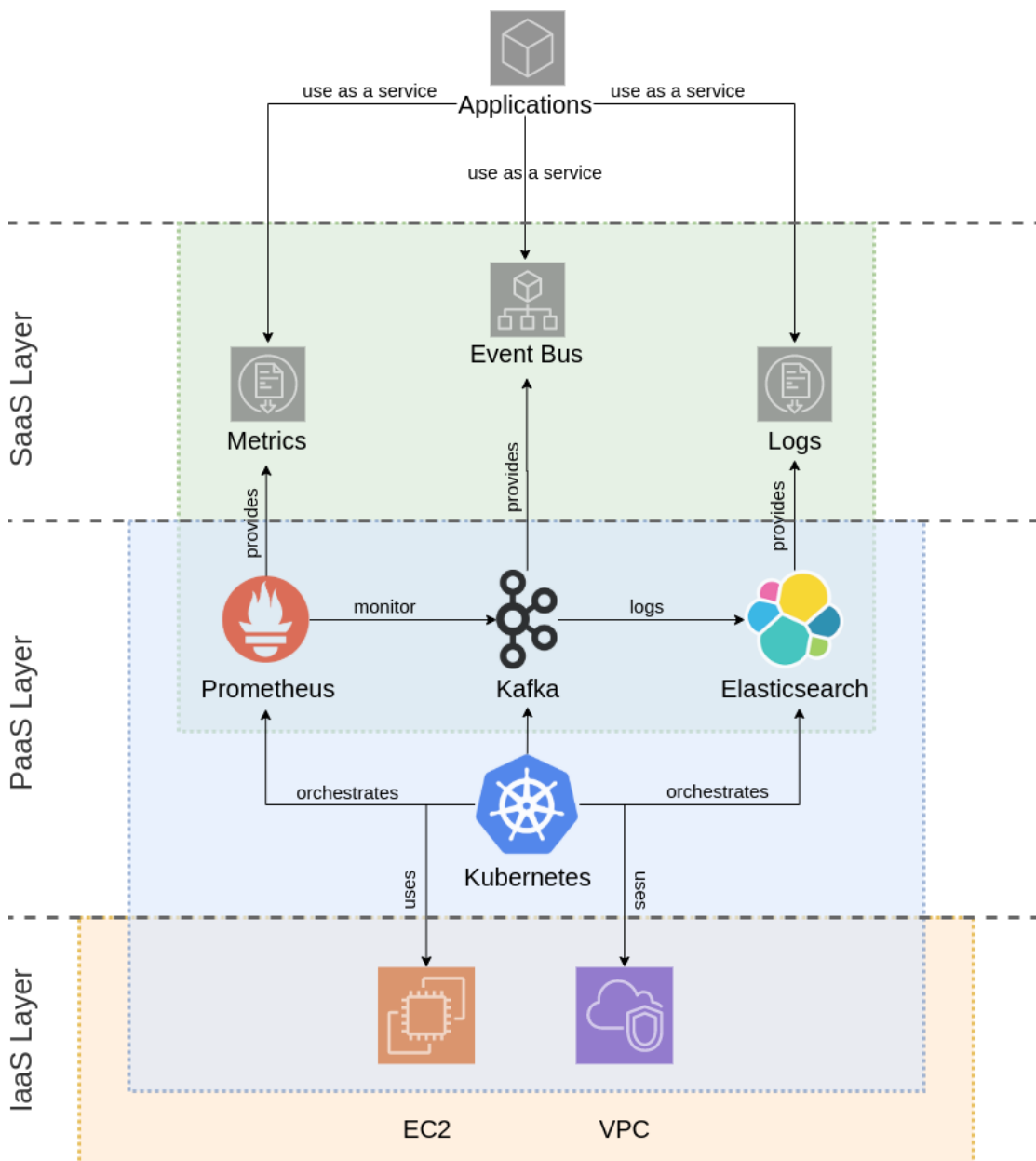
Let's assume an example to better explain the above statements: Resources defined at the IaaS layer means that they are managed at the Infrastructure level and that they offer functionalities as a service to the next layer which is the Platform. A virtual machine or a network router is fully managed at the Infrastructure level and if a resource from the Platform wants to leverage them, it just gets pure CPU, Memory, and Network. The way these offerings are administered or configured is totally agnostic to the resource that belongs to the Platform. The platform only knows that can reserve and use resources from Infrastructure as is. The same chain of usability applies and from Platform to Software Layer etc.

Our implementation's IaaS layer will include all the core components needed to start building on a higher abstraction level. These components involve virtual machines to provide compute resources, virtual routers and switches to provide networking, connectivity, and routing, and finally persistent storage.

At the PaaS layer, we will include the core components that have as the main dependency the offerings of the IaaS layer and that provide the functionalities needed from the SaaS layer to offer final solutions to end-users. Components included at this level are the Kubernetes Cluster as well as the Kafka and Monitoring Cluster.

Finally, at the SaaS layer, we meet our actual offerings which are (a) Event Bus as a Service which is backed by the Kafka Cluster in the Platform layer, and (b) the Metrics and Logs as a Service which are backed by our Monitoring Cluster in the Platform layer as well.

You can see an overview of all the three layers combination plus separation in the following diagram.



IaaS / PaaS / SaaS High Level Overview

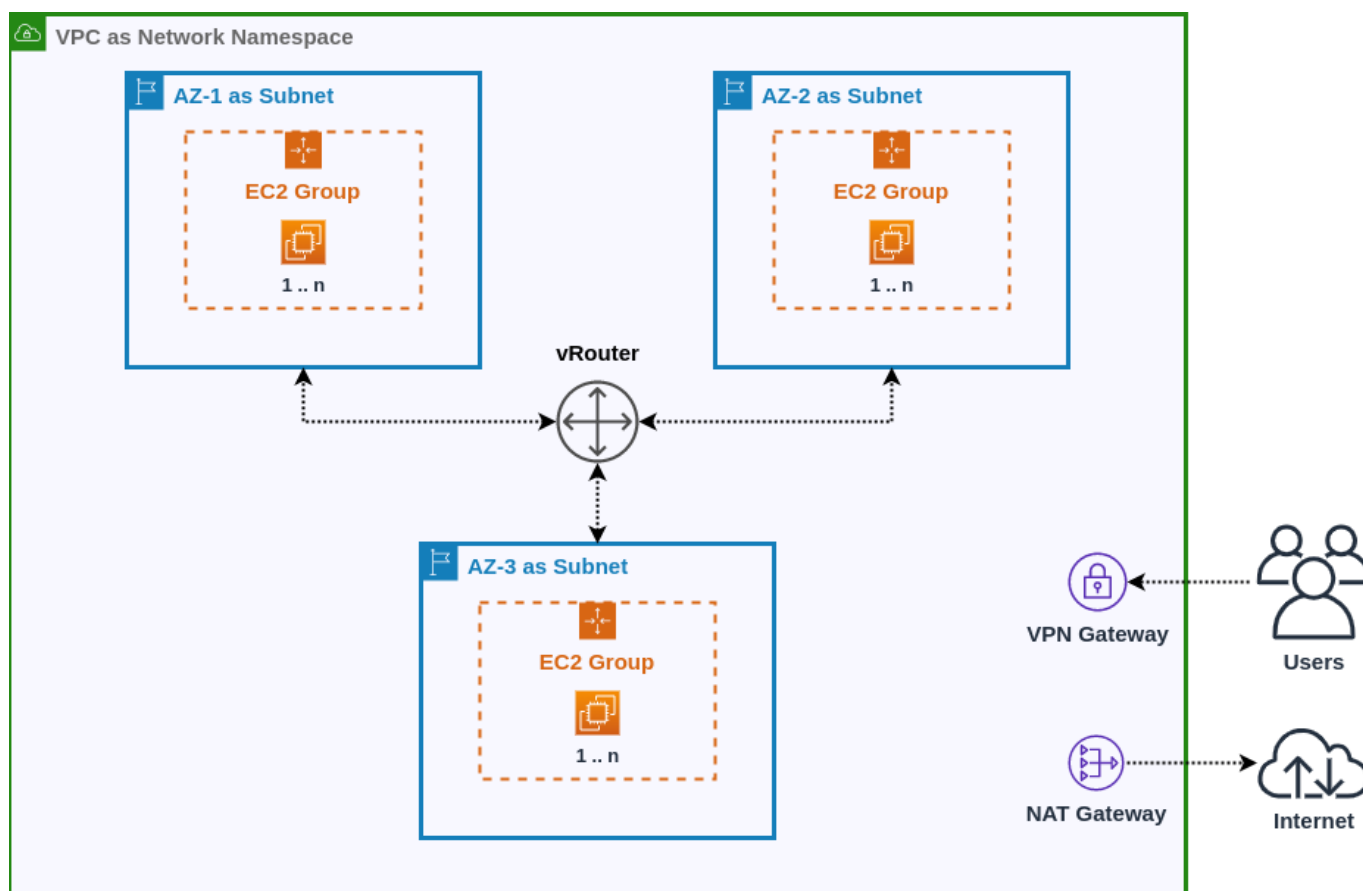
Each Layer will be thoroughly described in detail in the following sections.

3.3.1 IaaS Layer

This layer acts as our foundations; in simple terms, our Infrastructure. Even if we actually get these resources as virtualized components from a public cloud provider, we will design our system without that in mind.

We will focus on leveraging these resources to create an architecture that can offer the required Reliability, Resilience that can tolerate multiple physical locations failure. Remember, even if we consider all these as totally virtualized resources, in the end, it's an abstraction of a bare-metal computer hosted in a real Data Center.

Our implementation's design goal was to be applicable across different providers, from Cloud to Bare Metal ones.



IaaS Layer Overview

We will create a VPC to host all our compute and network resources. Reminder at this point, a VPC stands for Virtual Private Cloud and it is an abstraction that delivers an isolated and virtualized namespace for specific compute and network resources.

VPC will handle resources that will be spread across three different Availability Zones. Availability Zone acts as a separated physical location, in simple words a Data Center. Leveraging three different physical locations will ensure our need for a highly available distributed system that can tolerate failures even on a regional level.

The reason behind choosing three instead of two Availability Zones stands to the fact that the technologies we have chosen are - in their majority - using Raft Consensus Algorithm to reach Quorum [36] for taking decisions and elect leadership between their master nodes. As a result and as the Raft RFC suggests, we need at least three different nodes to avoid split brains and Platform inconsistencies [36].

In each Availability Zone, we will provision a Node Group, so-called from cloud provider as an Autoscaling Group. Each one of this Node Group will host the virtual machines needed to provide compute power for the software we will deploy at the next phase, in the PaaS Layer. This Node Group will also deliver an auto-scaling feature with a policy that triggers based on CPU or Memory Load. For example, if a Node Group that hosts a number of virtual machines has a specific CPU load for a specific time interval that operators had decided, a new virtual machine will be spawn to join this Node Group and deliver the extra compute power required.

Regarding networking needs for the above architecture, we need to ensure connectivity and transparency between our Nodes in different physical locations in the OSI Layer 3 and 4[ref]. With the word transparency we mean at all of our Nodes in the VPC will be under the same supernatted network, they will be assigned a unique IP to advertise themselves to communicate with other Nodes. In addition, Nodes inside

the VPC would have the ability to communicate with other Nodes without NAT.

That given, each Availability Zone will host a separate and unique in a manner of CIDR subnet that will be a subset of our VPC root subnet, that all the virtual machines will be assigned IP address from.

The connectivity between Availability Zones and the routing between different subnets will be handled by a virtual network function provided by a cloud provider that acts as a Virtual Router. This Virtual Router includes all the functionalities needed so our resources will be provided with a transparent underlay network to work on.

Virtual Router will also be also responsible for the policies regarding network traffic. The summary of the rules for our traffic in four points: (1) communication inside our private network is allowed from all to all by default (2) ingress traffic from external networks to our internal network is denied by default (3) ingress traffic from external networks from authorized users is allowed only through dedicated VPN Gateway (4) egress traffic to external networks is allowed from all to all by default.

As mentioned before, we want our internal resources to be totally isolated from public internet ingress access. Operators and End-Users who need to access internal resources will be able to do so by using a VPN Gateway that we will create. This VPN Gateway would be provided by a dedicated virtual machine inside our VPC that will be configured with a VPN server service and that will be the only node in our network that will be assigned with a public IP from our cloud provider besides the private one from our Virtual Router.

Internal resources that need egress to access public internet, will be able to do so by a NAT Gateway that will be associated with Virtual Router.

We want all our implementation aspects to be declared as Code so we will use Terraform [ref] to achieve that scope for the IaaS layer. Terraform is a tool for building, changing, and versioning Infrastructure as a Code [29].

```
resource "aws_vpc" "vpc" {
  cidr_block      = "${var.vpc_cidr}"
  enable_dns_hostnames = "true"

  tags {
    Name        = "${var.stack_name}"
    Stack       = "${var.stack_name}"
    ManagedBy   = "Terraform"
  }
}

1 references
variable "vpc_cidr" {}
2 references
variable "stack_name" {}
0 references
output "vpc_id" {
  value = "${aws_vpc.vpc.id}"
}
```

Terraform VPC Declaration Example

Actors of IaaS Layer

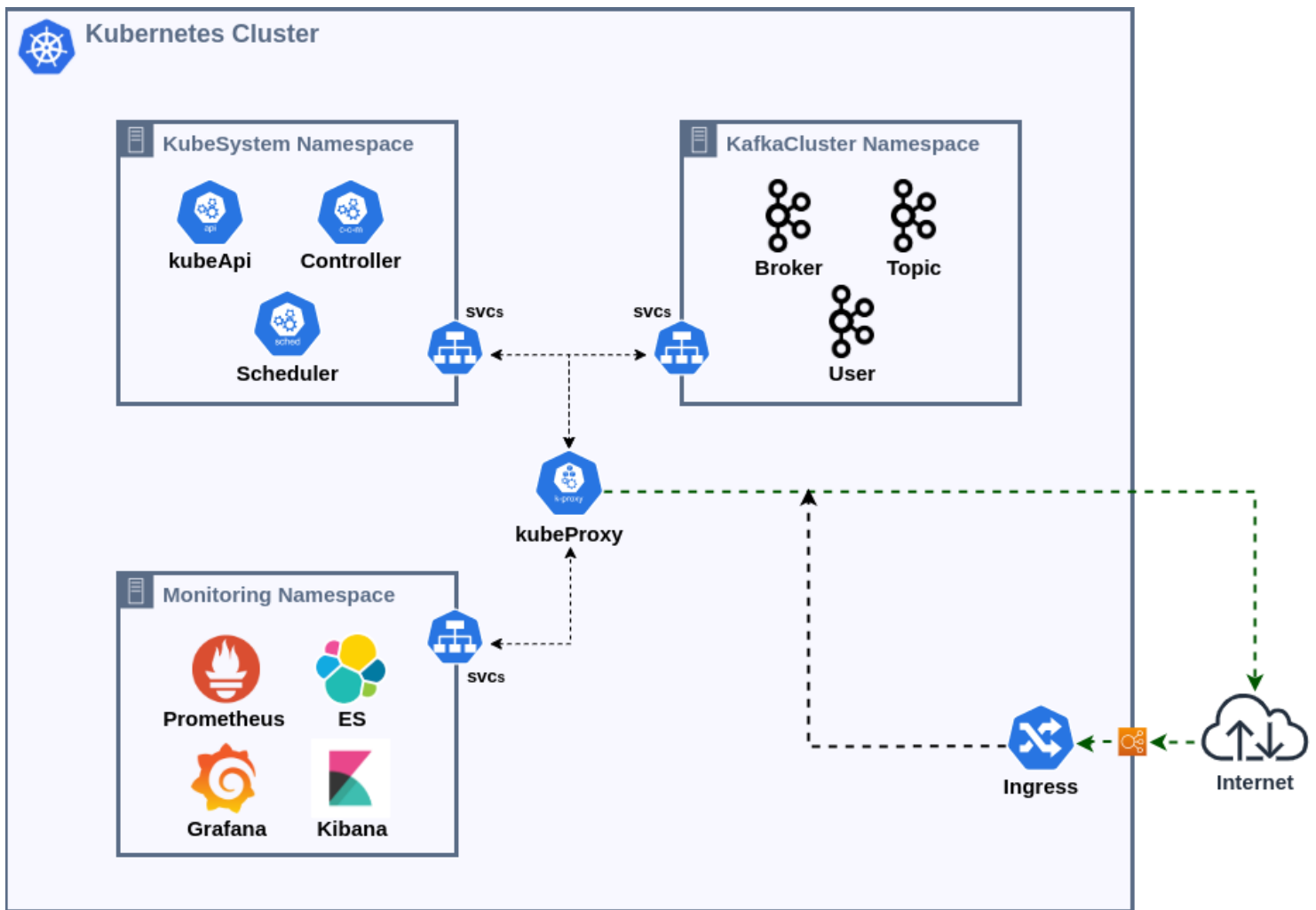
Platform Operators to configure and administer declared resources.

Summary of IaaS Layer Design Principles

- A. Multiple Availability Zones to ensure High Availability
- B. At least Two Physical Locations Fault Tolerance
- C. Transparent Underlay Network
- D. Nodes inside the VPC can communicate with without NAT
- E. Users can communicate with VPC only through VPN Gateway
- F. VPC can communicate with the outside world through NAT Gateway

3.3.2 PaaS Layer

This layer acts as the place where the promised offerings of our implementation are introduced. We will reserve infrastructure resources directly from the IaaS layer to deploy and configure our core runtime which is the main dependency to our SaaS layer.



PaaS Layer Overview

We will combine features of various great open source technologies mentioned in section 3.2, to produce a Platform that resolves all the dependencies needed from the SaaS layer to offer end users a fully managed containerized environment to deploy their apps with a

fully-managed Event Bus for communication and the essential Monitoring.

A Kubernetes Cluster will be created so as the rest components to be directly deployed in an environment that offers state declaration.

The Control plane of the Kubernetes cluster will be held by three Master Nodes. These nodes will be split among all Availability Zones -each physical location will host a master node- to ensure high availability for the whole cluster orchestration.

Kubernetes master nodes will hold the essentials services needed for cluster orchestration: (1) API server which is responsible for storing and validating cluster objects state (2) Controller which is responsible for keeping cluster and objects' state at the defined one (3) Scheduler which is responsible for leveraging Infrastructure resources in such a way to keep cluster state healthy (4) Etcd which is responsible to provide a reliable key-value store to API server [12]

Kubernetes Worker Nodes will also split among all Availability Zones, with the actual number of nodes to be decided from autoscaling group policies after measuring the compute resources requirements.

In each Kubernetes worker node, essential components for managing the node will be deployed: (1) Kubelet which is responsible for managing container processes (2) KubeProxy which is responsible for looping in iptables rules to ensure proper routing for Kubernetes overlay network and (3) Weave CNI which is responsible to provide container processes with virtual network interfaces so as to gain the ability to join Kubernetes overlay network. [12]

All above components will be held under Kubernetes Namespace with the name "Kube-System" and the scope of this namespace is to group all the services needed for proper Kubernetes cluster-based functionalities.

Two more Kubernetes Namespaces will be created: Kafka and Monitoring.

In the Kafka Namespace, we will deploy a Kafka and a Zookeeper cluster that will be managed from a CRD. Here, we will deploy all the resources needed - Pods, Persistent Volumes, etc - to provide the functionality of our Platform's Event Bus. A more detailed description of this namespace will follow in section 3.4.2.

In the Monitoring Namespace, we will deploy Prometheus, Grafana, Elasticsearch, and Kibana. Here we will deploy all the resources needed - Pods, Persistent Volumes, etc - to provide the functionality of our Platform's Monitoring System. A more detailed description of this namespace will follow in section 3.4.3.

All the above deployments are served with Guaranteed QoS which means that they will get priority and guaranteed to compute resources reservation from the IaaS layer.

At the same time, these deployments will be offered an Autoscaling Policy -where applicable- to ensure that if there is a need for more compute resources to get them.

We apply these policies as we consider components of the PaaS layer is critical ones for our implementation.

Actors of PaaS Layer

Platform Operators to configure and administer declared resources.

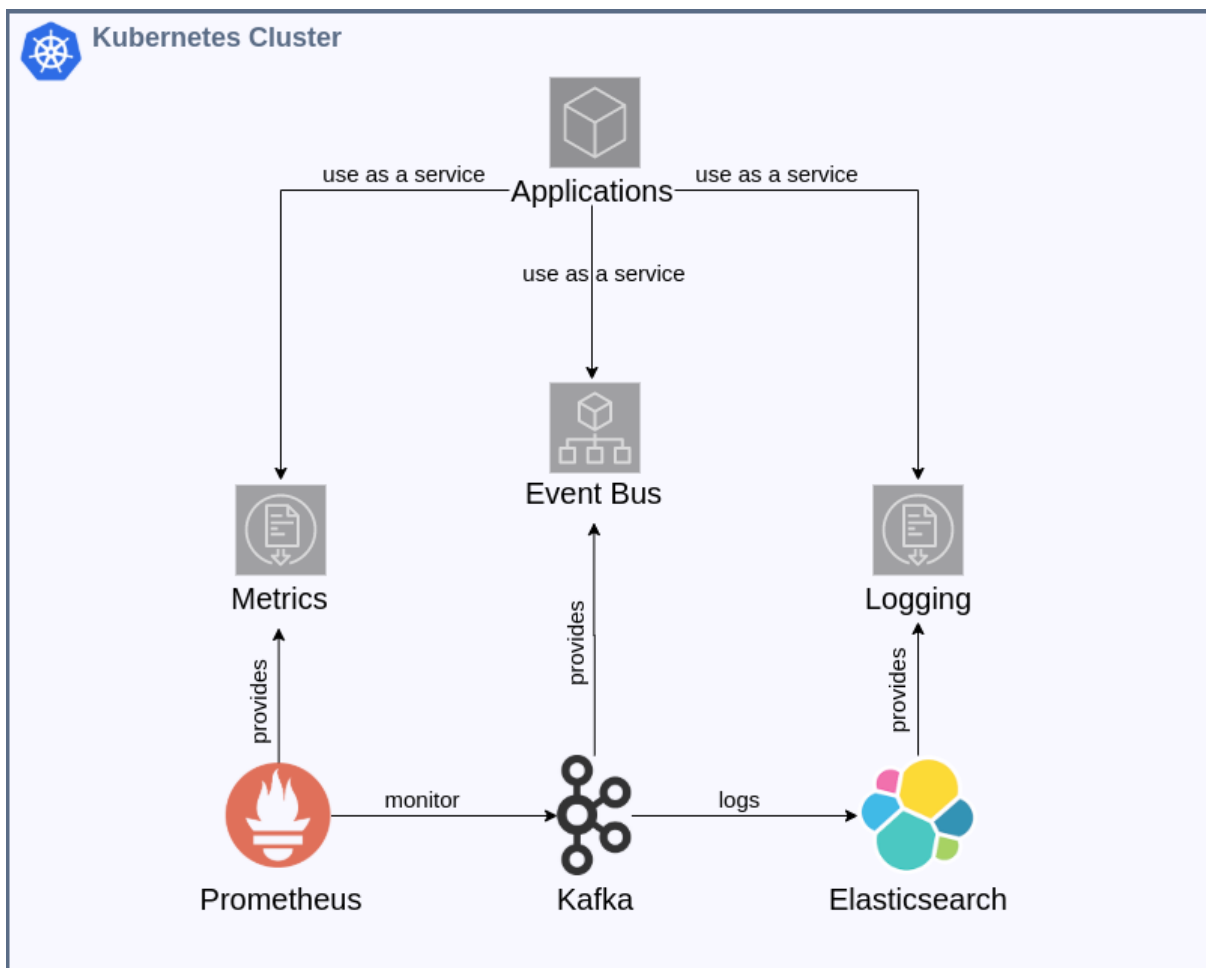
Summary of PaaS Layer Design Principles

- A. Kafka will provide through its dedicated operator, a robust self-healed cluster
- B. Elasticsearch will provide through its dedicated operator, a robust self-healed cluster
- C. Kubernetes System deployments will be served with Guaranteed QoS
- D. Kafka deployments will be served with Guaranteed QoS
- E. Monitoring deployments will be served with Guaranteed QoS
- F. Deployments will declare Horizontal Pod Autoscaling Policy to ensure High Availability
- G. Host network traffic will be shaped with kubeProxy
- H. Pods in the host network can communicate with all Pods without NAT

3.3.3 SaaS Layer

In this layer, we finally introduce our offerings to Software Engineers as end-users and to the Applications deployed in our Platform.

What we guarantee in this layer -by leveraging PaaS layer offerings- is that end-user(s) can develop and deploy Application(s) that (1) Kubernetes will manage their desired state (2) Event Bus will be present for communication between Applications and (3) Metrics and Logging will be provided upon request.



SaaS Layer Overview

All above functionalities can be enabled directly from the application's deployment manifest that Kubernetes handles.

For a producer Application, Kafka dependencies can be declared by involving the following manifest in their deployment:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic
metadata:
  name: topic_name
  labels:
    strimzi.io/cluster: cluster_name
spec:
  partitions: 4
  replicas: 2
  config:
    retention.ms: 86400
    retention.bytes: -1
    segment.bytes: 1073741824
    cleanup.policy: delete
    min.insync.replicas: 1
    unclean.leader.election.enable: true
```

Kafka Topic Declaration at Application Manifests

For a consumer Application, Kafka dependencies can be declared using environmental variables:

```
env:
  {{- range $key, $value := .Values.env }}
  - name: {{ $key }}
    value: "{{ $value }}"
  {{- end }}
  - name: KAFKA_TOPIC
    value: topic_name
```

Kafka Topic Declaration at Application Manifests through ENV variable

For an Application that wants Prometheus to collect its metrics, the following annotations would enable the scraping:

```
annotations:  
  kubernetes.io/change-cause: "{{ .Values.changeCause }}"  
  {{ if .Values.monitoring.enabled -}}  
  prometheus.io/scrape: "true"  
  prometheus.io/path: "{{ .Values.monitoring.path }}"  
  prometheus.io/port: "{{ .Values.monitoring.port }}"  
  {{- end }}
```

Prometheus Scraping Declaration at Application Manifests

Actors of SaaS Layer

- **Applications** deployed in the Platform that asks guarantees for the offerings
- **Software Engineers** as end-users and Stakeholders of the offerings

Summary of SaaS Layer Design Principles

- A. Kubernetes will handle the desired state of Deployments in the layer of runtime, autoscaling, and self-healing
- B. Kafka will provide through its dedicated operator, a robust self-healed cluster
- C. Prometheus is able to scrape every Deployment is requested to
- D. Producer Applications can declare their Kafka dependencies inside their Manifests
- E. Consumer Application can declare their Kafka dependencies inside their Manifests
- F. Fluentd will deliver all stdout/err of the Platform at Elasticsearch

3.4 Specifications

3.4.1 Kubernetes Cluster

As we mentioned before, Kubernetes will be deployed in virtual machines across all three Availability Zones to ensure high availability and quorum requirements.

	Replicas	Multi-AZ	CPU req	Memory req
Master Nodes ASG	3	Yes	8 vCores	16 GB
Worker Nodes ASG	3 .. n	Yes	8 vCores	16 GB


Kubernetes Cluster underlying Infrastructure Specifications

All Kubernetes Nodes will be delivered with Debian as an operating system as it offers great flexibility in tuning Kernel and Network-specific configurations that are essential for an optimized host node.

Components that will be deployed along with Kubernetes Cluster and that will be responsible for core functionalities are the following:

- **Traefik:** An Ingress controller that is responsible for delivering network traffic from outside the cluster to internal resources. [37, 38]
- **CoreDNS:** A DNS service provider for Kubernetes Clusters [39]
- **MetricsServer:** Get cluster-wide metrics directly from kubelet and Kernel cadvisor module to provide Kubernetes objects monitoring [40]
- **Autoscaler:** A component that leverages metrics exposed from MetricsServer and communicates with IaaS API to manage the underlying virtual machines group scaling operations. Autoscaler guides IaaS to spawn or absorb worker nodes. [41]

- **FluentD:** Daemon that scrapes all the cluster's Pods stdout and stderr to deliver these streams to an Elasticsearch cluster for monitoring purposes [43]
- **Weave:** CNI that attaches to all the cluster's Pods a virtual network interface so they can join and communicate through the Kubernetes overlay network [44]

	Function	Type	Repl cas	CPU req	Memory req
Traefik	Ingress	Deployment	3 .. 6	300m	128 Mi
CoreDNS	DNS Service	Deployment	3 .. 6	300m	256 Mi
Autoscaler	Cluster Nodes AutoScaling	Deployment	2 .. 3	100m	128 Mi
MetricsServer	Cluster Monitoring	Deployment	2 .. 3	200m	128 Mi
Kube2IAM	IAM Roles Attach	DaemonSet	n	100m	128 Mi
FluentD	Logs Forwarding	DaemonSet	n	200m	128 Mi
Weave	CNI	DaemonSet	n	200m	128 Mi

[Core Components Overview Specifications](#)

We want every aspect of our Platform to be defined as a Code in order to be maintainable, idempotent across different infrastructure environments, avoid or restrict manual operations and reduce complexity for Operators.

To meet the above requirements, Kubernetes Cluster will be provisioned and configured by Kops and its core components will be deployed with Helm.

Kops is a utility that takes as input a Kubernetes Cluster declaration as manifest and produces as output an idempotent deployment of it. By communicating with IaaS layer API, Kops asks for needed resources and when they are ready, it applies the Kubernetes configuration supplied. [30]

```
apiVersion: kops.k8s.io/v1alpha2
kind: Cluster
metadata:
  creationTimestamp: null
  name: __CLUSTER__
spec:
  api:
    loadBalancer:
      type: Internal
  authorization:
    rbac: {}
  channel: stable
  cloudProvider: aws
  configBase: s3:bucket-location
  etcdClusters:
  - etcdMembers:
    - encryptedVolume: true
      instanceGroup: master-__REGION__
      name: main
  - etcdMembers:
    - encryptedVolume: true
      instanceGroup: master-__REGION__
      name: events
  iam:
    allowContainerRegistry: true
    legacy: false
  kubeAPIServer:
    oidcClientID: __STACK__-cluster
    oidcGroupsClaim: groups
    oidcIssuerURL: oidc_url
    oidcUsernameClaim: email
```

Kops Cluster Manifest Example

Helm is an application package manager for Kubernetes. With Helm, you can define even the most complex manifest leveraging GoLang templates. In addition, Helm keeps versioning of each deployments' revision manifests. [31]

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app.kubernetes.io/name: {{ .Release.Name }}
  template:
    metadata:
      annotations:
        {{ if .Values.monitoring.enabled -}}
        prometheus.io/scrape: "true"
        prometheus.io/port: "{{ .Values.monitoring.port }}"
        {{- end }}
    spec:
      containers:
        - name: {{ .Release.Name }}
          image: "{{ .Values.image.registry }}{{ .Values.image.name }}:{{ .Values.image.tag }}"
          ports:
            - name: metrics
              containerPort: 9308
              protocol: TCP
          env:
            {{- range $key, $value := .Values.env }}
            - name: {{ $key }}
              value: "{{ $value }}"
            {{- end }}
          resources:
            {{ toYaml .Values.resources | indent 12 }}
            {{- with .Values.nodeSelector }}
            nodeSelector:
            {{ toYaml . | indent 8 }}
            {{- end }}
            {{- with .Values.affinity }}
            affinity:
            {{ toYaml . | indent 8 }}
            {{- end }}
            {{- with .Values.tolerations }}
            tolerations:
            {{ toYaml . | indent 8 }}
            {{- end }}
```

Helm Deployment Template Example


3.4.2 Kafka Namespace

Kafka Namespace is the place where we will group our Kubernetes objects required for Kafka Cluster deployment (Pods, Services, Persistent Volumes, etc).

Strimzi Operator is an open-source project that gives the ability to provision and configures Kafka Clusters as a Code and as Kubernetes manifest. [46]

We will leverage Strimzi Operator [46] features as it can handle Kafka operations in a more reliable way than a straight Kubernetes deployment. The reason for such a diverse between Strimzi and plain Deployment is that Strimzi as a custom resource definition is developed to understand and intervene in Kafka's application logic rather than manipulating Kubernetes process-based attributes. The operator will be deployed as CRD. [45]

Kafka Cluster and Zookeeper Cluster (which acts as a key-value store and is a hard dependency for a Kafka) will be deployed as StatefulSets and will be managed by Strimzi Operator.

	Function	Type	Repl cas	CPU req	Memory req
Strimzi	Operator	CRD	1	300m	128 Mi
Kafka	Broker	StatefulSet	3 .. n	1000m	256 Mi
Zookeeper	KV Store	StatefulSet	3	200m	128 Mi

[Kafka Namespace underlying Kubernetes Specifications](#)

As we mentioned before, we want to deliver all aspects of our implementation as a code and Strimzi offers a variety of nice features for this.

With Strimzi, we can declare a Kafka Cluster as Kubernetes manifest or we can even declare a Kafka Topic as Kubernetes manifest with reference to an existing Kafka Cluster. Let's see some examples of the above cases described.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: unipi
spec:
  kafka:
    version: 2.4.0
    replicas: 6
    listeners:
      plain: {}
      external:
        type: loadbalancer
        overrides:
          bootstrap:
            dnsAnnotations:
              external-dns.alpha.kubernetes.io/hostname: kafka-k8s.unipi.thebeat.co
              external-dns.alpha.kubernetes.io/ttl: "60"
  config:
    offsets.topic.replication.factor: 3
    transaction.state.log.replication.factor: 3
    transaction.state.log.min.isr: 2
    auto.create.topics.enable: true
    log.message.format.version: "2.4"
  rack:
    topologyKey: failure-domain.beta.kubernetes.io/zone
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 10Gi
        deleteClaim: false
```

Kafka Cluster as Kubernetes Manifest Example

As you can observe, with Strimzi we can define both system-level specifications like the size of the block device that will be mounted for Kafka to hold its data and Kafka's application-level specifications like rack awareness topology and broker configuration, etc.

Our implementation's Kafka Cluster will be provisioned with at least three nodes to ensure High Availability and brokers will be enforced through their rack awareness policy, to be deployed across all Availability Zones, one at a time in a round-robin way.

Furthermore, we can declare a Kafka Topic as Kubernetes Manifest providing both reference to an existing Kafka Cluster and application-level topic configuration.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic
metadata:
  name: unipi-test
  labels:
    strimzi.io/cluster: unipi
spec:
  partitions: 10
  replicas: 2
  config:
    retention.ms: 46400
    retention.bytes: 1073741824
    segment.bytes: 107374182
    cleanup.policy: delete
    min.insync.replicas: 1
    unclean.leader.election.enable: true
```

Kafka Topic as Kubernetes Manifest Example

Topics will be enforced to hold at least three Replicas and at least one in-sync Replica to leverage the previously mentioned policy.

3.4.3 Monitoring Namespace

Monitoring Namespace is the place where we will group all our Kubernetes objects required for Elasticsearch, Prometheus, Grafana, Kibana, and Node Exporter deployments (Pods, Services, Persistent Volumes, etc).


ECK Operator is an open-source project that gives the ability to provision and configures Elasticsearch and Kibana Clusters as a Code and as Kubernetes manifests [47].

We will leverage ECK Operator features as it can handle Elastic Stack operations in a more reliable way than a straight Kubernetes deployment. The reason for such a diverse between ECK and plain Deployment is that ECK as a custom resource definition is developed to understand and intervene in Elastic Stack application logic rather than manipulating Kubernetes process-based attributes. The operator will be deployed as CRD. [45]

Elasticsearch and Kibana Cluster will be deployed as StatefulSet and will be managed by their corresponding Operator which will be deployed as CRD

Prometheus and Grafana will be deployed as StatefulSet and Deployment accordingly.

Node Exporter, which acts as a monitoring agent at the IaaS layer, will be deployed as DaemonSet so to ensure its presence on each one of the virtual machines which act as Kubernetes worker nodes.

	Function	Type	Repl cas	CPU req	Memory req
Elasticsearch	Log Store	StatefulSet	3	300m	128 Mi
Kibana	Log Visualize	StatefulSet	1	500m	256 Mi
Prometheus	Metric Store	StatefulSet	1	200m	128 Mi
Grafana	Metric Visualize	Deployment	1		
Node Exporter	System Monitoring	DaemonSet	n		

Monitoring Namespace underlying Kubernetes Specifications

As we mentioned before, we want to deliver all aspects of our implementation as a code, and ECK besides Helm will provide this for Monitoring Namespace.

With ECK, we can declare an Elasticsearch Cluster as Kubernetes manifest.

We can define both system-level specifications like the size of the block device that will be mounted for Elasticsearch to hold its data and Elastic Stack application level specifications like node role and xpack security configuration etc.

Our implementation's Elasticsearch Cluster will be provisioned with at least three nodes to ensure High Availability and nodes will be enforced through their rack awareness policy, to be deployed across all Availability Zones, one at a time in a round-robin way.

Let's see an example of the Elastic Stack declaration.

```
apiVersion: elasticsearch.k8s.elastic.co/v1
kind: Elasticsearch
metadata:
  name: unipi
spec:
  version: 7.5.2
  updateStrategy:
    changeBudget:
      maxSurge: -1
      maxUnavailable: 1
  http:
    tls:
      selfSignedCertificate:
        disabled: true
  nodeSets:
  - name: unipi
    count: 3
    config:
      node.master: true
      node.data: true
      node.ingest: true
      node.store.allow_mmap: false
      xpack.security.authc:
        anonymous:
          username: anonymous
          roles: superuser
          authz_exception: false
  volumeClaimTemplates:
  - metadata:
      name: elasticsearch-data
    spec:
      accessModes:
      - ReadWriteOnce
      resources:
        requests:
          storage: 30Gi
      storageClassName: gp2
```

Elasticsearch Cluster as Kubernetes Manifest Example

Prometheus and Grafana will be provisioned as simple Kubernetes StatefulSet as their single-node architecture introduces limitations in the functionality extensibility that a Custom Resource Definition could enhance them with.

4. Results

4.1 Unit Test

Scope

In this section, we want to validate that the functionalities of the offerings on our Platform are working end-to-end. The results of this test could provide us with useful insights, on a first iteration basis, to validate our **Functionality Claims** that users should expect from our Platform.

Scenarios and Acceptance

We will simulate all the Use Cases mentioned in previous sections, to make certain verifications that the defined offerings delivered proper functionality. Scenarios will be held under normal Load conditions for the whole Platform. By normal, we mean about ~65-70% resource utilization.

Acceptance comes when functionality for each scenario meets our actual expectations.

Topology

We will deploy various dummy applications that will produce and consume messages from multiple topics to ensure our Kafka as a Service offering. At the same time we will be monitoring these applications and our Platform as well through the Monitoring system we have already deployed.

Applications that act as Producers will publish to specific topics and applications which act as Consumers that subscribe to specific topics.

Output

All Unit test Scenarios **passed successfully**.

- Scenario: Prometheus Scraping

The platform successfully scraped metrics, for every deployment annotated its request to utilize Prometheus.

- Scenario: Elasticsearch Logging

The platform successfully pushed logs to no-SQL documentDB, for every deployment that was exposing its logs to stdout and stderr.

- Scenario: Kafka Dependencies Declaration

The platform successfully created the dependencies needed from both publisher and subscriber applications. Kafka Topic creation, modification, and deletion were tested multiple times and for different kinds of deployments and the results always validated proper functionality.

- Scenario: Kafka Scaling Operations

The platform successfully provisioned scaling operations on the Kafka cluster upon modifications on the corresponding manifest resource. One thing to consider is that the Kafka cluster scaled in a manner of a number of broker nodes, but added or removed nodes were not fully utilized from Kafka as the rebalancing phase was not triggered at that point automatically.

We have tracked this as a feature that should be considered as future work and that could be implemented by adding Cruise Control functionality in our Platform, see more in section 5.

#	Scenario	Result	Notes
1	Prometheus Scraping	Pass	
2	Elasticsearch Logging	Pass	
2	Producer Applications can declare their Kafka dependencies inside their Manifests	Pass	
3	Consumer Application can declare their Kafka dependencies inside their Manifests as environmental variables	Pass	
4	Scale-Out Kafka Cluster	Pass	Manual cluster rebalance was needed for new brokers to be fully utilized.
5	Scale In Kafka Cluster	Pass	Manual cluster rebalance was needed for old brokers to not be included in cluster state

[Unit Test Scenarios Output Overview](#)

4.2 Load Test

Scope

In this section, we want to calculate the performance level our Platform will deliver per CPU core. The results of this test could provide us with useful insights as a first iteration basis to declare **Performance Claims** that users should expect from our Platform's Kafka Cluster.

Scenario and Acceptance

We will create the appropriate conditions to measure end-to-end latency of how many messages could a Kafka pipeline deliver under an acceptable time threshold and utilizing CPU resources near to one vCore per broker.

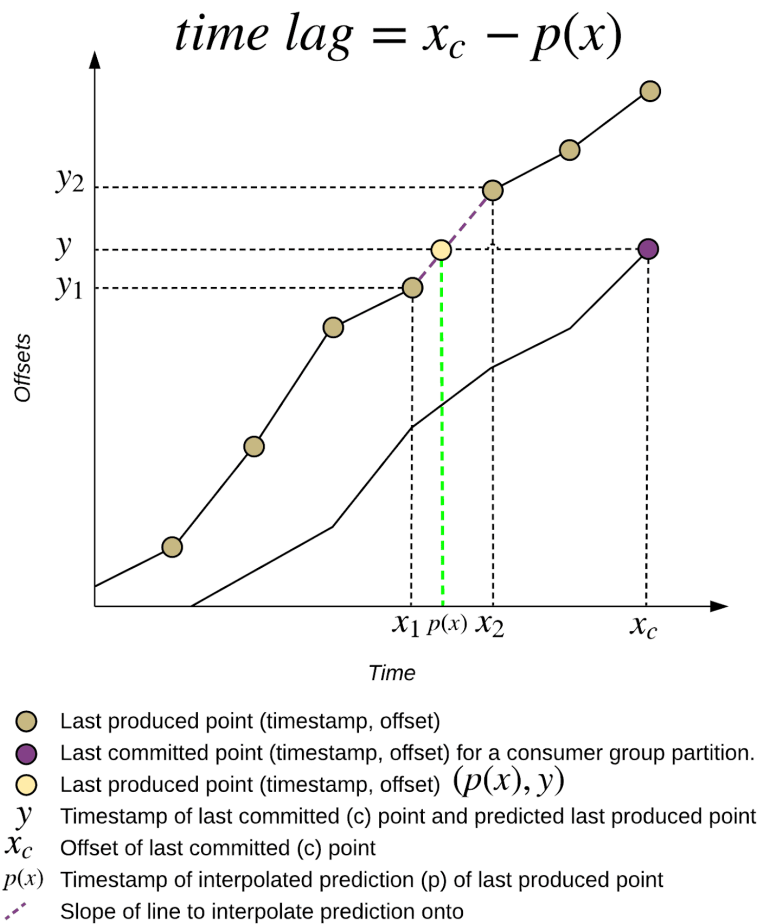
Key	Value
Broker Count	6
CPU Req Target	~ 1000m
Client Count	1 .. 50
Message Count	10k .. 1m per client
Message Rate (rps)	100 .. 250 per client
Message Size (bytes)	512 / 1024 / 4096
Replication Factor	3
Min InSync Replicas	1
ACK Config	-1
Retention Policy	7d or 5GiB
Consumer Lag Target	< 10000 ms

In order to measure our ConsumerLag target indicator, we will use a utility called Kafka Lag Exporter [55] that bind to a cluster and calculates the latency for a topic per consumer group, by processing the following formula:

Linear Interpolation Formula
(solved for x)

$$p(x) = x_2 - (y_2 - y) \frac{(x_2 - x_1)}{(y_2 - y_1)}$$

Time Lag Estimate



Consumer Lag Calculation Formula

Acceptance come to the point where our pipeline reaches ~1 vCore and simultaneously ConsumerLag is under 10000 ms.

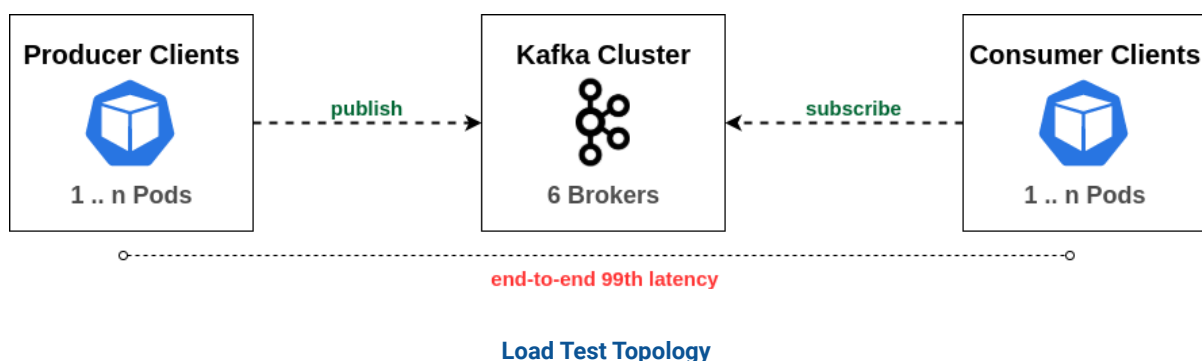
We are choosing to specify a threshold for end-to-end latency to keep an acceptable time interval for generic pipelines to respect. If we measure one core utilization with enormous lag times, the load test would not provide us with results that are useful for production environments.

Topology

X number of clients will act as Producers that will publish to a specific Topic and Y number of clients will act as Consumers that subscribe to a specific Topic.

Kafka brokers replicas count will be set to six and replication factor for topics will set to three. Active partitions will be set to ten.

The producer's ACK configuration will be set to -1 to ensure the delivery of a message is acknowledged to be replicated to the desired copies so our cluster can handle node failures.

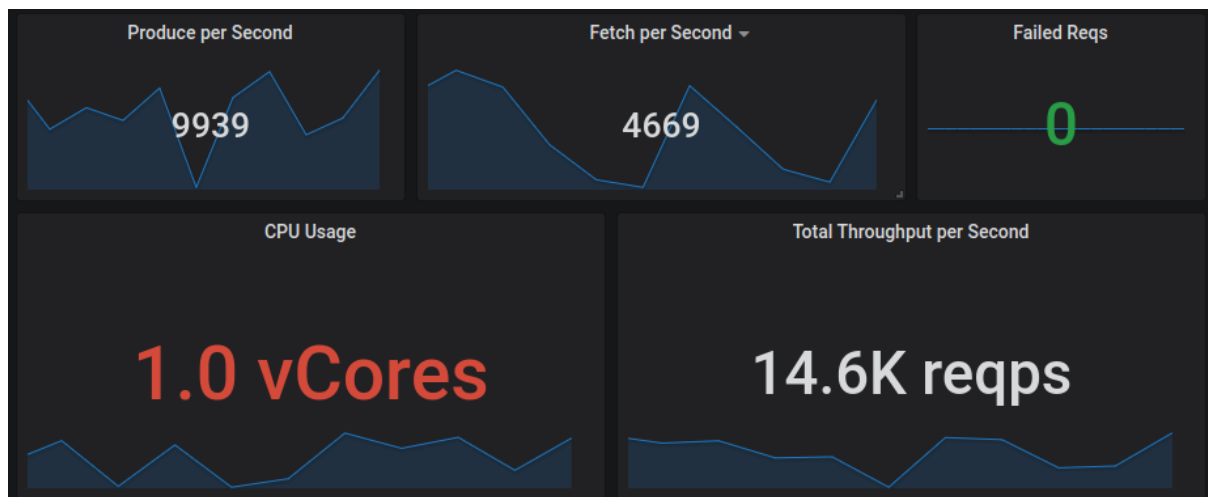


Output

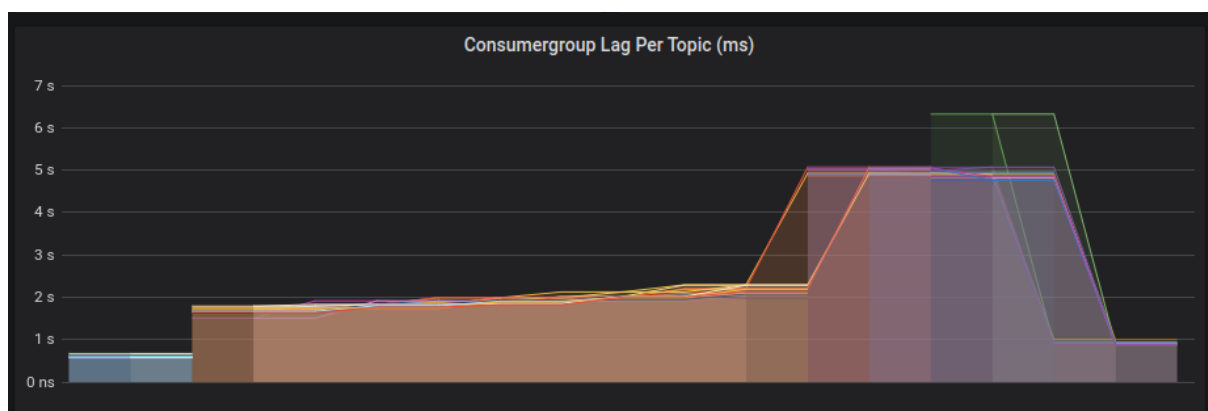
After many iterations of testing with different values for environment configuration, we reach our target to utilize one vCore with **80** clients as Producers and **10** clients acting as Consumers.

The amount of the producing size was adjusted accordingly to bring the essential traffic to reach the targeted compute load. Consuming side was static at ten clients, to meet the actual number of partitions as if we had a greater number of consumers, coordination lag may introduce at our group.

Our Platform, without any kind of optimization on specific pipeline needs, can deliver a throughput of **~14600 req/s per CPU core**.



Load Test Target Reached Iteration: CPU Usage and Throughput Graph



Load Test Target Reached Iteration: Consumer Lag Graph

4.3 Stress Test

Scope

In this section, we want to validate that our Platform continues to operate normally under specific disaster scenarios. We will involve testing beyond normal operational capacity, often to a breaking point, to observe its behavior.

The results of this kind of test could provide us with useful insights, on a first iteration basis, to validate our **Reliability Claims** that users should expect from our Platform's Kafka Cluster.

Scenarios, Target Indicator and Acceptance

We will simulate specific disaster scenarios, to measure the cluster and pipeline disturbance introduced at the functionality of the defined offering during the time of each incident.

We will test (a) Heavy Traffic Spikes (b) Broker Failure © Kubernetes Node failure and (d) Rolling Upgrade of Kafka Cluster

Scenarios will be triggered during a stabilized state of the Platform running under normal Load conditions. By normal, we mean about ~65-70% resource utilization.

The disturbance will be measured by monitoring the possible increase rate of the Consumer Lag metric. The formula for measuring Consumer Lag has described in section 4.2 where we implement the Load Test

Acceptance criteria deliver success when the disturbance is < **10000 ms**.

Topology

Several dummy applications will act as Producers that will publish to specific topics and several dummy applications will act as Consumers that subscribe to specific topics when stress scenarios will be held.

Output

All Stress Test Scenarios **passed Successfully**.

- Scenario: Broker Failure

The platform delivered exciting results in case of a Kafka Broker failure. Disturbance to applications could not even be observable in our pipeline end-to-end latency formula with a granularity of 30 seconds between scraping.

Examining applications side to find a way to measure the impact of broker failure, we came up that disturbance was equal to the time needed for the consumer group to trigger and finish its rebalance phase which is actually under **~100 milliseconds** and the time needed for Broker recovery was about **~30 seconds** which is actually the time needed for a new Pod for Kafka StatefulSet to be scheduled.

- Scenario: Node Failure

The platform delivered the same results as in the above scenario of Kafka Broker failure as for Kafka, a corrupted Kubernetes Nodes is translated as the Pods that Node was holding and were acting as Kafka Brokers that turned into a fail state.

The time needed for Node recovery was about **~120 seconds**. During the time with a corrupted Kubernetes node, Disturbance for applications utilizing Kafka Cluster was under **~100 milliseconds** in total.

- Scenario: Rolling Upgrade

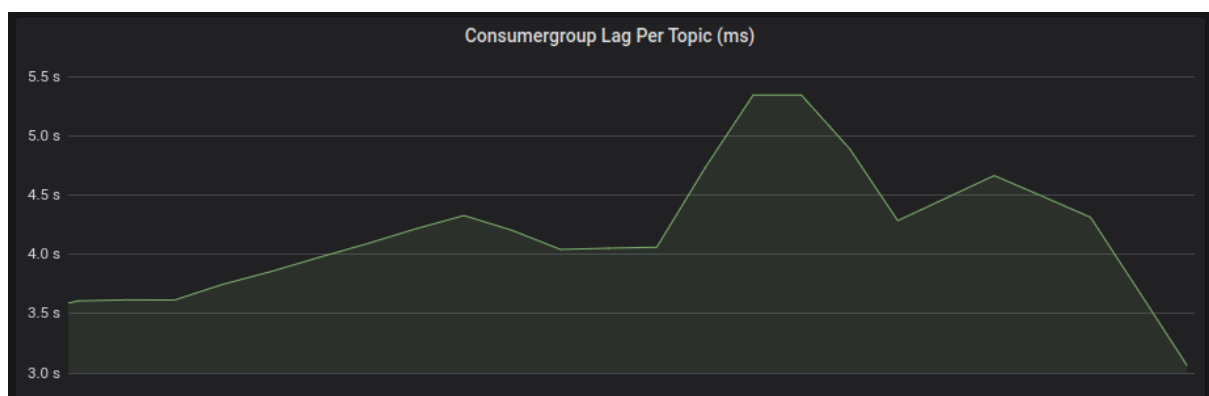
Modification on Kafka Brokers and then rolling apply the new configuration was a smooth and non-blocking operation that held and tested several times. Nothing more to report here except that disturbance for applications was under **~100 milliseconds** after each Broker restart.

- Scenario: Heavy Traffic Spike

In this scenario, we simulate a load spike by throttling a huge amount of traffic to the cluster. What we observed is an increase in consumer lag by **~50%** and that Kafka chose to give priority to produce throughput rather than the consumer side.

We also got some flapping in the status of Kafka Controller between healthy and not healthy state but cluster continued to deliver proper functionality. The indicator with the highest increase was the time spent during the memory garbage collection stage, something normal as Kafka had to cope with a huge number of requests to handle.

The scenario was successful as the disturbance did not violate our acceptance threshold.



Stress Test Load Spike Disturbance

Another observation to consider is that during the simulation of the heavy traffic spike scenario we indirectly validated our results during Load Test that was held in section 4.2.

Putting so much pressure on the cluster which was translated as throughput increase, we came out to verify that cluster was robust enough not to stop working but at the same time cluster could not cope to deliver throughput greater than 15000 msg/s per core. That number is in sync with the results for the pipeline threshold we also got during load testing Kafka to measure its performance per core.

#	Scenario	Result	Disturbance
1	Kafka Broker Failure	Pass	< 100 ms
2	Kubernetes Node Failure	Pass	< 100 ms
3	Kafka Cluster Rolling Upgrade	Pass	< 80 ms
4	Trigger a heavy Traffic Spike	Pass	< 3000 ms

[Stress Test Scenarios Output Overview](#)

5. Future Work

In this section, we describe possible extra Features that can be enabled on our Platform

Service Level Operator

Service level operator abstracts and automates the service level of Kubernetes applications by generation SLI & SLOs to be consumed easily by dashboards and alerts and allow that the SLI/SLO's live with the application flow.

This operator interacts with Kubernetes using the CRDs as a way to define application service levels and generate output service level metrics.

Although this operator is thought to interact with different backends and generate different output backends, at this moment only uses Prometheus as input and output backend.

Repo: <https://github.com/spotahome/service-level-operator>

Cruise Control

Cruise Control is a general-purpose system that continually monitors Kafka clusters and automatically adjusts the resources allocated to them to meet predefined performance goals. In essence, users specify goals, Cruise Control monitors for violations of these goals, analyzes the existing workload on the cluster, and automatically executes administrative operations to satisfy those goals.

Repo: <https://github.com/linkedin/cruise-control>

Cloud Events Specification

Events are everywhere. However, event producers tend to describe events differently.

The lack of a common way of describing events means developers must constantly re-learn how to consume events. This also limits the potential for libraries, tooling, and infrastructure to aid the delivery of event data across environments, like SDKs, event routers, or tracing systems. The portability and productivity we can achieve from event data are hindered overall.

CloudEvents is a specification for describing event data in common formats to provide interoperability across services, platforms, and systems.

Repo: <https://github.com/cloudevents/spec>

Schema Registry

Schema Registry provides a serving layer for your metadata. It provides a RESTful interface for storing and retrieving Avro schemas. It stores a versioned history of all schemas, provides multiple compatibility settings, and allows the evolution of schemas according to the configured compatibility setting. It provides serializers that plug into Kafka clients that handle schema storage and retrieval for Kafka messages that are sent in the Avro format.

Repo: <https://github.com/confluentinc/schema-registry>

7. References

- [1] Martin Kleppmann. Designing Data-Intensive Applications. O'Reilly Media, 2017
- [2] Sam Newman. Building Microservices. O'Reilly Media, 2015
- [3] Alessandra Levcovitz, Ricardo Terra, Marco Tulio Valente. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. <https://arxiv.org/pdf/1605.03175.pdf>, 2016
- [4] David Byrne, Carol Corrado, Daniel Sichel. The Rise of Cloud Computing. <https://www.imf.org/~media/Files/Conferences/2017-stats-forum/session-6-byrne.ashx>, 2017
- [5] Kim-Kwang Raymond Choo. Cloud computing: Challenges and future directions. <https://aic.gov.au/file/6229/download?token=mY1RSeBw>, 2010
- [6] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara Fabrizio Montesi, Ruslan Mustafin, Larisa Safina. Microservices: yesterday, today, and tomorrow. <https://arxiv.org/pdf/1606.04036.pdf>, 2017
- [7] Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. <http://www.archive.ece.cmu.edu/~ece845/docs/containers.pdf>, 2014
- [8] Cisco, Redhat. Linux Containers: Why They're in Your Future and What Has to Happen First. <https://pdfs.semanticscholar.org/8ad5/000af66f60772645bec3e7e9eaf14acde7e5.pdf>, 2014
- [9] Aaron Grattafiori. Understanding and Hardening Linux Containers. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-1-1.pdf, 2016
- [10] Maria Rodriguez, Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. <http://www.buyya.com/papers/CloudContainerOrchSPE.pdf>, 2018
- [11] Rajkumar Buyya, Maria Rodriguez, Adel Nadjaran, Jaeman Park. Cost-Efficient Orchestration of Containers in Clouds: A Vision, Architectural Elements, and Future Directions. <https://arxiv.org/pdf/1807.03578.pdf>, 2018
- [12] Kubernetes Documentation. <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>
- [13] Leila Abdollahi, Mohamed Aymen Saied, Maria Toeroe, Ferhat Khendek. Kubernetes as an Availability Manager for Microservice Applications. <https://arxiv.org/pdf/1901.04946.pdf>, 2019
- [14] Kubernetes Standardized Glossary. <https://kubernetes.io/docs/reference/glossary/?fundamental=true>
- [15] Kubernetes Terminology. <https://betterprogramming.pub/kubernetes-a-detailed-example-of-deployment-of-a-stateful-application-de3de33c8632>

- [16] Brendan Burns, Brian Grant, David Oppermeiher, Eric Brewer, John Wilkes. Borg, Omega, and Kubernetes. <https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44843.pdf>, 2016
- [17] David Oppenheimer, Eric Tune, John Wilkes. Large-scale cluster management at Google with Borg. <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/43438.pdf>, 2015
- [18] Using Kubernetes: Lessons after a year in Production. <https://techbeacon.com/devops/one-year-using-kubernetes-production-lessons-learned>
- [19] Kafka Official Documentation. <https://kafka.apache.org/documentation/>
- [20] Jay Kreps, Neha Narkhede, Jun Rao. Kafka: a Distributed Messaging System for Log Processing. <http://notes.stephenholiday.com/Kafka.pdf>, 2014
- [21] Martin Kleppmann, Jay Kreps. Kafka, Samza and the Unix Philosophy of Distributed Data. <https://martin.kleppmann.com/papers/kafka-debull15.pdf>, 2015
- [22] Butler Lampson. Hints and Principles for Computer System Design. <https://www.microsoft.com/en-us/research/uploads/prod/2019/09/Hints-137-short.pdf>, 2019
- [23] AWS Official Documentation. <https://docs.aws.amazon.com/>
- [24] Linux Debian Documentation. <https://www.debian.org/doc/>
- [25] Prometheus Overview. <https://prometheus.io/docs/introduction/overview/>
- [26] Elasticsearch Overview. <https://towardsdatascience.com/an-overview-on-elasticsearch-and-its-usage-e26df1d1d24a>
- [27] Jatin Varlyani. What is Git and how to use it. <https://levelup.gitconnected.com/what-is-git-how-to-use-it-why-to-use-it-explained-in-depth-76a5066abaaa>, 2019
- [28] Docker Official Documentation. <https://docs.docker.com/>
- [29] Terraform Official Documentation. <https://www.terraform.io/docs/index.html>
- [30] Kops Official Git Repo. <https://github.com/kubernetes/kops/tree/master/docs>
- [31] Helm Official Git Repo. <https://github.com/helm/helm>
- [32] Jenkins Official Git Repo. <https://github.com/jenkinsci/jenkins>
- [33] Using Grafana with Prometheus. <https://grafana.com/docs/grafana/latest/features/datasources/prometheus/>
- [34] Kibana Official Documentation. <https://www.elastic.co/guide/en/kibana/current/introduction.html>
- [35] Eugene Gorelik. Cloud Computing Models. <http://web.mit.edu/smadnick/www/wp/2013-01.pdf>, 2013
- [36] Heidi Howard. Analysis of Raft Consensus. <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-857.pdf>, 2014

- [37] What is Kubernetes Ingress?
<https://kubernetes.io/docs/concepts/services-networking/ingress/>
- [38] Traefik Ingress Official Documentation. <https://docs.traefik.io/>
- [39] CoreDNS Official Git Repo. <https://github.com/coredns/coredns>
- [40] Metrics Server Official Git Repo.
<https://github.com/kubernetes-sigs/metrics-server>
- [41] Cluster Autoscaler Official Git Repo. <https://github.com/kubernetes/autoscaler>
- [42] Kube2IAM Official Git Repo. <https://github.com/jtblin/kube2iam>
- [43] FluentD Architecture. <https://www.fluentd.org/architecture>
- [44] CNI for Docker Containers, with Weave & Calico.
<https://www.weave.works/blog/cni-for-docker-containers/>
- [45] Extend the Kubernetes API with CustomResourceDefinitions.
<https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/>
- [46] Strimzi Operator Overview. <https://strimzi.io/>
- [47] ECK Operator Official Git Repo. <https://github.com/elastic/cloud-on-k8s>
- [48] Comparison of instruction set architectures.
https://en.wikipedia.org/wiki/Comparison_of_instruction_set_architectures
- [49] Instruction Set Architectures Overview.
https://en.wikipedia.org/wiki/Instruction_set_architecture
- [50] Event-driven architecture.
https://en.wikipedia.org/wiki/Event-driven_architecture
- [51] Observer pattern. https://en.wikipedia.org/wiki/Observer_pattern
- [52] Event-Driven Architecture Implementation.
<https://medium.com/hackernoon/event-driven-architecture-implementation-140c51820845>
- [53] Database Shard Overview.
[https://en.wikipedia.org/wiki/Shard_\(database_architecture\)](https://en.wikipedia.org/wiki/Shard_(database_architecture))
- [54] Optimizing Kafka.
<https://www.confluent.io/wp-content/uploads/Optimizing-Your-Apache-Kafka-Deployment-1.pdf>
- [55] Kafka Lag Exporter Official Git Repo.
<https://github.com/lightbend/kafka-lag-exporter#estimate-consumer-group-time-lag>
- [56] Varghese B, Buyya R. Next-generation cloud computing: New trends and research directions.
https://pureadmin.qub.ac.uk/ws/portalfiles/portal/134629726/paper_v2.pdf, 2017
- [58] Extending Kubernetes API with CRDs
<https://medium.com/velotio-perspectives/extending-kubernetes-apis-with-custom-resource-definitions-crds-139c99ed3477>