B. Comp. Dissertation

# Resource Orchestration for a Cloud Native Stream Processing Engine

By

Yin Ruohang

Department of Computer Science

School of Computing

National University of Singapore

2023/2024

B. Comp. Dissertation

# Resource Orchestration for a Cloud Native Stream Processing Engine

By

Yin Ruohang

Department of Computer Science

School of Computing

National University of Singapore

2023/2024

Project No: H159500

Advisor: Richard Ma Tianbai

Deliverables:

FYP Report

Executor code

Kubernetes Operator code

# Abstract

The transition of computing resource management in businesses towards cloud environments underscores the need for enhanced efficiency, scalability, and resilience in real time data processing applications on the cloud with current existing solutions coming short in one or more of these areas. Addressing this need, this paper introduces Invoker, a Stream Processing Engine (SPE) that is designed and built on cloud-native principles from the ground up. At the heart of our approach is the strategic use of Kubernetes, which serves as a foundational component of cloud infrastructure, to enable a robust cloud-native architecture for stream processing. By designing Invoker to function as Pods within Kubernetes, we tap into the platform's advanced resource orchestration capabilities through the development of custom Kubernetes Operators. This architecture not only supports dynamic reconfiguration but also ensures scalable processing performance. Our evaluation through the deployment of a streaming application demonstrates the functionality and adaptability of this approach. The outcomes reveal a fully operational prototype that heralds the next generation of cloud-native SPEs, showcasing the effectiveness of our design and opening pathways for further advancements in the domain.

Subject Descriptors

C.2.1 Multiple Data Stream Architectures (Multiprocessors)

C.2.4 Distributed Systems

Keywords:

Cloud Native Systems, Stream Processing, Kubernetes Operators, Resource Orchestration, Distributed Systems

Implementation Software and Hardware:

Java 1.8, Go 1.21, Kubernetes v1.27, Kafka v3.7.0, Docker v24.0.7, KinD v0.20.0, OperatorSDK v1.33.0, MicroK8s v1.29

# Acknowledgements

I would like to thank my advisor, Professor Richard Ma, for the continuous support and guidance throughout the project. I would also like to thank Yancan for his continuous involvement in designing this system throughout and his constructive feedback for improving as an academic researcher.

# Contents

# 1 Introduction

This paper details the design and implementation of my work on Invoker: A Cloud Native Stream Processing Engine a project led by Professor Richard, Ma Tianbai and PhD student Mao Yancan. The work I have done for my Final Year Project involves implementing the runtime of this system at an application level by implementing the execution Pods on Kubernetes as well as designing and implementing the resource orchestration layer with a custom Kubernetes Operator. In this paper, I will first explore the project and motivation, give some background information on the existing systems, and discuss Kubernetes as a container orchestration platform. Then, I will explain the system requirements of such a system, how I designed and implemented the resource orchestration layer, and conduct testing using a stateful stream processing application.

## 1.1 Stream Processing Engine

Stream processing engines (SPEs) are specialized software systems designed for handling continuous streams of data in real time. They enable the collection, processing, analysis, and integration of large volumes of fast-moving data. (Fragkoulis, Carbone, Kalavri, & Katsifodimos, 2023) This capability is crucial for applications and services that require immediate insights and responses based on incoming data. As the scale of data processing increases in the computing industry, stream processing has become a key component in various domains such as financial services, telecommunications, online retail, IoT (Internet of Things), and real time analytics. A stream processing engine helps developers write these applications by providing a framework to develop on. The user will usually define the input and output data flow, define a topology of processing the data, and use the libraries given by the platform to create a stream processing application. Some popular stream processing engines in the industry include Apache Flink (*Apache Flink®*, 2023), Apache Spark (Zaharia et al., 2016), Kafka Streams (Apache, 2023), and Ray (Moritz et al., 2018). The most important features outside of functionality a stream processing system offers include performance, state management, fault tolerance and high availability, elasticity, scalability, and reconfiguration (Fragkoulis et al., 2023). Here I will explain some of these requirements and what is needed to fulfill them.

- High throughput and low latency: SPEs are optimized to process a high volume of data events with minimal delay, ensuring that data analysis and decision-making

are as real time as possible.

- State Management: SPEs can maintain and manage state information across data streams, which is crucial for complex event processing where the outcome depends on past and present events.
- Fault Tolerance: SPEs provide mechanisms to handle failures gracefully, ensuring that data processing can continue uninterrupted in the event of hardware or software failures.
- Elasticity: They are able to ensure result correctness under fluctuating workloads and increase parallelism to prevent performance degradation.
- Scalability: They can scale horizontally to manage increased loads, allowing more data to be processed in parallel without sacrificing performance.
- Reconfiguration/Flexibility: They can be configured when deployed, allowing them to adapt to different workload and change the manner of handling data without having to be restarted.

There are many existing systems with solutions and different strategies implemented to solve these problems, including heartbeat, watermark, partition of storage, exactly-once processing, etc, and extensive research surveys have been done on them. However, the currently widely used stream processing engines are designed and built with the mind of running in traditional bare metal machine servers with cloud integration added on later in the development of a project. Thus, they are unable to fully utilize the features of a cloud resource orchestration platform, and fall short of fulfilling all the requirements of flexibility, scalability, and reliability in a cloud environment. The goal of the overarching project, Invoker, is to build a stream processing engine in a cloud native manner. By starting with designing the system from a cloud native perspective, we aim to leverage the benefits of the cloud to be able to fulfill the shortcomings of existing stream processing engines.

## 1.2 Cloud Native

As defined by the Cloud Native Computing Foundation (CNCF), which coined the term in 2015 alongside the introduction of Kubernetes, Cloud Native refers to the approach of building and running applications that fully exploit the advantages of cloud computing

environments (CNCF, 2015). Primarily, this means applications are designed to run in a cloud environment, leveraging the cloud's flexibility, scalability, and resilience.

One of the foremost benefits of cloud native design is its inherent scalability and flexibility. Cloud native applications should be built in a way that allows them to scale out or in seamlessly based on demand. (Gannon, Barga, & Sundaresan, 2017) This is a significant departure from traditional architectures that often require significant effort and downtime to scale. By utilizing services like Kubernetes, applications can automatically adjust their resources, ensuring optimal performance even under varying loads. This not only improves user experience but also optimizes resource usage, leading to cost efficiencies. By leveraging the cloud's pay-as-you-go pricing model, organizations can significantly reduce resource operational costs through using cloud native systems.

Cloud native applications are also designed to be resilient in the face of failures. By adopting microservices architectures and containerization, applications can isolate failures to a small component without affecting the entire system. This design principle ensures that the application can provide continuous service even when parts of it are experiencing problems, also contributing to cost reductions.

The benefits of cloud native architecture discussed align closely with the requirements of stream processing engines, with the ability to scale dynamically and process data in real time being critical to a SPE. Cloud native design also facilitates the construction of applications that can efficiently process large volumes of data, providing insights with minimal latency.

# 2 Study of Existing Systems

## 2.1 Stream Processing Engines

In this section, I will briefly discuss the existing stream processing systems of Apache Flink and Apache Kafka Streams, two of the more mainstream stream processing engines, and give a brief overview of their architecture as well as their efforts to adapt to the cloud environment.

### 2.1.1 Apache Flink

Apache Flink is a powerful, open-source stream processing framework for stateful computations over unbounded and bounded data streams. Its core runtime is a distributed dataflow graph, which represents the operations and data streams of an application. The graph consists of vertices (operations) and edges (data flows). (*Apache Flink*®, 2023) Flink applications are automatically parallelized, with each operation being executed in multiple instances across a cluster of machines for scalability. The system is fault-tolerant, using distributed snapshotting for state consistency and recovery. Flink is designed to run on various cluster managers like YARN, Mesos, Kubernetes, or standalone on bare metal machines. A Flink application is composed of a series of operations or transformations (like map, filter, reduce) which are executed in a distributed manner. These operations are broken down into tasks that can be executed concurrently.

Flink's architecture consists of a JobManager and multiple TaskManagers following a master worker pattern, where the JobManager assigns tasks to run on each TaskManager. The developer would communicate with the JobManager by writing a stream processing application to submit as a job to the JobManager via an REST interface. The JobManager is the central coordinator in the Apache Flink architecture. It is responsible for managing the life cycle of jobs, including job scheduling, resource allocation, and fault tolerance. It also maintains a global view of all tasks and ensures they are executed according to the plan.

The TaskManager, on the other hand, is responsible for the execution of each task assigned by the JobManager. It can have multiple parallel instances while also having parallel tasks running in each TaskManager. For stateful operations, TaskManagers handle local state storage, ensuring that state is maintained across processing events. This state can be checkpointed and restored from the JobManager's coordination for fault tol-

erance. These two components communicate frequently to ensure smooth execution of Flink jobs. This includes the initial distribution of the job and its tasks, dynamic adjustments to processing (e.g., scaling out tasks), managing checkpoints for state, and handling failures.

Apache Flink's efforts into cloud native deployment come firstly in the form of seamlessly using Kubernetes as a resource provider with the support of Native Kubernetes. Running Flink in Native Kubernetes mode allows the JobManager to use Kubernetes APIs to directly request resources to dynamically allocate and deallocate resources to TaskManagers while the user interface remains the same. In Application mode, it directly containerizes the Job and TaskManagers into a Docker image and runs it on Kubernetes, while session mode separates the deployments to allow users to dynamically submit jobs on demand. Flink's open-source community has also created a Kubernetes Operator project, in which I will delve into in detail in the later sections.

### 2.1.2 Kafka Streams

Apache Kafka was originally not a stream processing engine, and served more as a highly performant and scalable event processing engine, acting as a message queue between services. It follows a PubSub model where producers send in messages to a topic and consumers read from it. Its parallelism is handled using multiple partitions for each topic, separating them into different data flows. However, open-source Apache Kafka also provides the Kafka Streams library to allow developers to integrate stream processing workflows directly into their Kafka cluster. (Apache, 2023) Compared to Flink, Kafka Streams is a much more lightweight library, but also lacks the full deployment lifecycle features Flink provides. Nevertheless, it is still a worthwhile case study to discover how stream processing applications are designed at a higher level.

Kafka Streams allows developers to define their stream processing logic in terms of a processing topology. The processing topology in Kafka Streams is executed by stream threads. Each thread can handle one or more tasks, where a task is the smallest unit of processing work that can be executed independently. These tasks are parallelizable units of work that correspond to topic partitions as shown in Fig. 1. This parallelism model allows Kafka Streams applications to scale horizontally by adding more stream threads, either within the same instance or across multiple instances. This scalability is crucial for handling large volumes of data efficiently. Kafka Streams leverages Kafka's

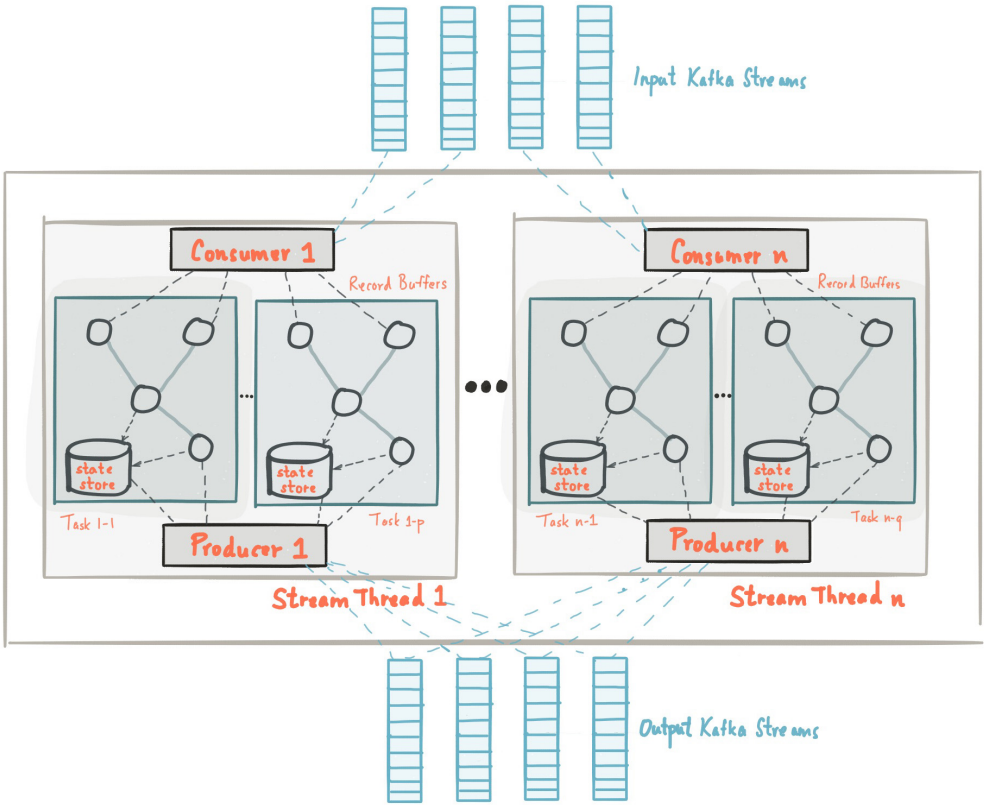


Figure 1: KafkaStreams architecture from (Apache, 2023)

consumer groups and partition assignment protocol to dynamically distribute stream processing tasks among available instances, facilitating load balancing and fault tolerance. An important feature of Kafka Streams is its support for stateful operations. Each task can have an associated local state store, enabling operations that require knowledge of previous data, such as windowed aggregations or joins. These state stores can be backed by Kafka topics, making the state fault-tolerant and recoverable in the event of a failure.

When working with Kafka Streams, the developer is responsible for defining the processing topology using the Kafka Streams API, managing application resources, and orchestrating deployment. The framework provides a high level of control to developers, allowing for custom stream processing logic tailored to specific requirements. Deployment of Kafka Streams applications, however, is less optimized for cloud-native environments compared to some other streaming frameworks like Apache Flink. The primary method for deploying Kafka Streams applications in a cloud environment is to containerize the application as a Deployment or StatefulSet and run it on a Kubernetes cluster. (Shapira & Sax, 2018) This approach leverages Kubernetes for managing the application's lifecycle, scaling, and resource allocation, but it may require additional effort from developers to optimize for cloud deployment, given that Kafka Streams does not provide as many

cloud-native features or optimizations out of the box.

## 2.2 Kubernetes

As much as this project's main goal is to build a stream processing engine, Kubernetes' container orchestration abilities perform an important role in the motivation as a foundational building block of Cloud Native systems. At its core, containers offer a lightweight, efficient method of resource isolation within a Linux environment, enabling applications to run in isolated user spaces while sharing the same operating system kernel. This isolation ensures that applications can be packaged with their dependencies and configurations, making them portable across different computing environments. Kubernetes enhances this container technology by providing a framework that automates the deployment, scaling, and management of containerized applications, ensuring they operate efficiently and reliably.

Kubernetes, also sometimes referred to as K8s, is based on the concept of resources. For a Kubernetes cluster running a whole system, the most basic resource is a Pod with containers running on it, serving as the basic unit of computation in the system. (*Kubernetes Components*, 2023) Each resource is defined by a YAML file manifest specifying its configuration and fields specific to that resource. For instance, the YAML manifest for Pod specifies the container runtime the containers in the Pod are running in, as well as which Docker images the Pod runs inside the containers. In addition to Pods, Deployments and Services are examples of higher-level resources that are commonly used to manage Pods. Deployments manage a set number of Pods running the same application, ensuring they are all healthy and functional, and if a Pod managed by the Deployment becomes unhealthy or fails, the Deployment is able to identify the failure and react accordingly. On the other hand, Service defines a policy to access a set of Pods, usually via a network. Other high level Kubernetes resources include PersistentVolume, ConfigMap, and StatefulSets, each providing features that can be useful in running a cloud computing application, depending on the circumstances. While these are logical components known to the Kubernetes cluster, the physical resources such as memory and CPU they consume distributed through the concept of Nodes. Nodes can be physical or virtual machines and is the basic unit of physical computation in Kubernetes. Each of them contains a Kubelet agent that manages the Node and the resources on the node by watching and updating them. One of the most important features of Kubernetes is being able to abstract this

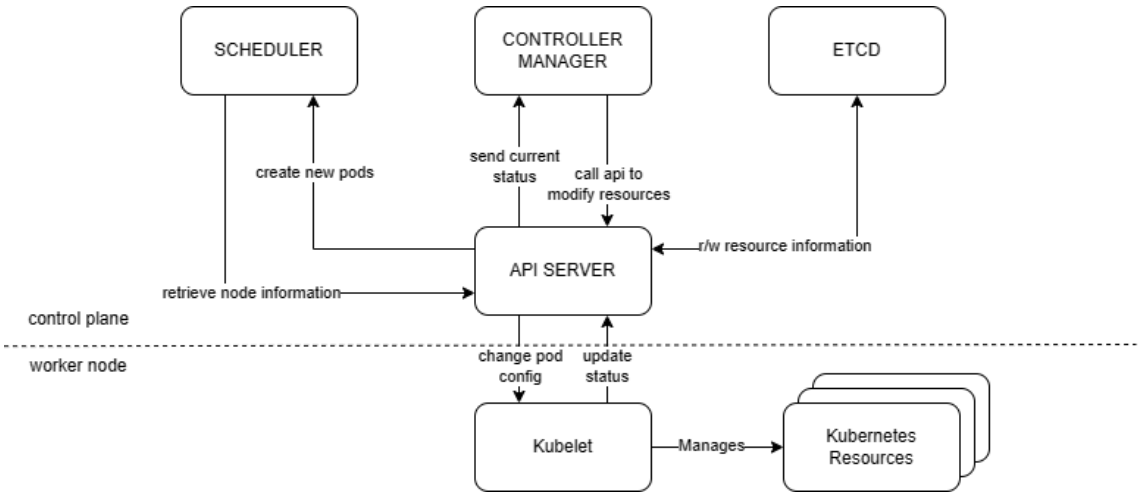


Figure 2: Kubernetes Control Plane's internal interactions and interactions with Worker Node components

knowledge of physical Nodes away so that developers deploying applications only need to care about creating a Deployment or Pod they want to run (Sayfan, 2023), thereby simplifying the need to understand the low level environment an application instance is running on.

At the high level, Kubernetes can be split into two separate planes. The logic to manage both the logical components and physical resources is all in the Kubernetes control plane, usually running on a separate Node. On the other hand, worker nodes are where the logic computation and storage occur. The control plane can be extended, but the most essential logic includes the api-server, etcd, controller-manager, and scheduler (*Kubernetes Components*, 2023). Figure 2 shows the control plane in relation to other Kubernetes Nodes and components. The api-server acts as the communication gateway between any components in Kubernetes as well as the developer. Requests to update or read resources pass from the api-server into each Node via the Kubelet, while resource status updates and metric information can be passed from the Pod to the api-server. The etcd is a consistent key-value storage used by Kubernetes to store all the cluster data, configuration data, and metadata for all the resources as a single source of truth. For example, for a Pod, its specifications, such as the container environment, will be stored and retrieved from the etcd. The scheduler allocates newly created Pods to a Node in the cluster based on a specific set of rules and algorithms. This is the main way that Kubernetes translates physical resources to logical Pod environments, and is how resource orchestration occurs in Kubernetes. Lastly, the controller manager is a compiled binary of a set of controllers for each high-level resource. In Kubernetes, each resource type, otherwise known as kind, has a

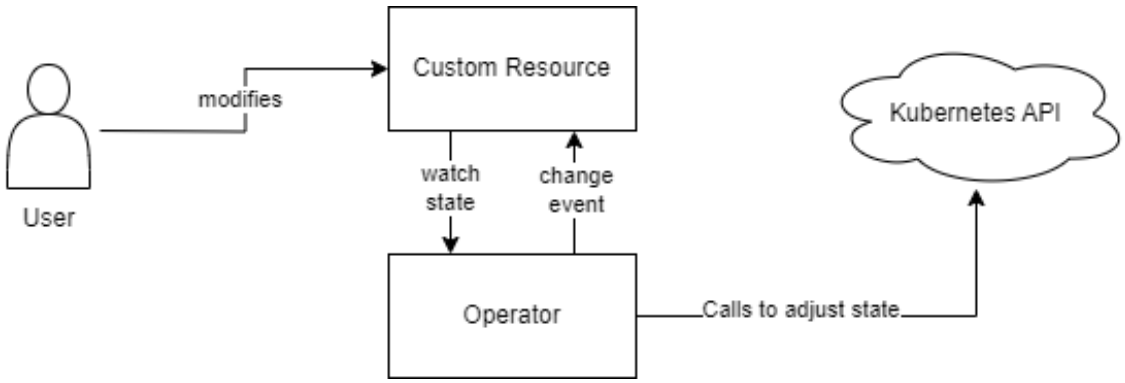


Figure 3: Kubernetes Operator reconciliation loop

controller deciding what to do and how to manage that resource. For example, the logic to spin up a new Pod when a Deployment's current number of Pod replicas does not match its required number is stored in the Deployment controller in the controller-manager. By calling various APIs to update resources in the cluster, controllers hold the logic that allows Kubernetes to automatically manage its containers. When running a new application on a Kubernetes cluster, this feature allows users to only have to create the YAML manifest of the resources and apply it to the cluster, leaving the creation and management of these resources entirely to Kubernetes, making the process a simple declarative approach of running an application.

### 2.2.1 Kubernetes Operators

The Kubernetes central logic and the control plane can be extended in many ways. To automate the deployment process of a complex application with many moving parts in a declarative approach, CustomResources (CRs) can be used. CustomResources can be used as built-in Kubernetes resources and take advantage of the mature Kubernetes API to control (Hausenblas & Schimanski, 2019). They are defined as objects by a CustomResourceDefinition (CRD) and are often used with custom controllers that watch the resources and perform operations based on the resource states. For example, to represent a database on Kubernetes, one can create a database CR containing information on its version, storage size, etc. The controller can then watch this resource and update the Kubernetes environment to match the current state of the cluster with the information on the CR by calling Kubernetes native APIs to, for instance, create a Deployment to host the storage and a Service to expose the database endpoint. This CR and controller pattern is known as the Operator pattern, where the custom resource defines a state, and the controller will try to match the current infrastructure with this state, making the deployment method native

to Kubernetes. One example of such a Kubernetes Operator is Fission, a framework on Kubernetes for serverless functions. Using a Function resource, Fission controllers will observe the Function definition and create and deploy the correct resources depending on the fields of the object using Kubernetes APIs (*Function.fission.io/v1*, 2023). Rather than running individual containers with applications that manage resources on FaaS, Fission performs resource orchestration in a very cloud native manner, allowing it to perform better and be easier to manage when ingrained into a Kubernetes ecosystem.

The custom controller aspect of Kubernetes operators is an application flow of logic running in the cluster. Just like built-in controllers inside the controller-manager, custom controllers operate in the same manner. Each controller watches one main resource it is responsible for, usually, this means that each CRD will have its dedicated controller the same way each built-in resource has its dedicated controller in the controller-manager. A custom resource can have secondary resources, or references, which should also be watched for changes. This relationship can be seen in the Deployment resource which owns a ReplicaSet resource and monitors its status. When watching a resource, the controller will have to check for any changes in its state as well as the state of the secondary resources. If the current state of the resources running on Kubernetes differs from the described information in the CR according to the developer, the controller must reconcile the state by calling various CRUD (Create, Read, Update, Delete) APIs to Kubernetes on these resources in order to match the existing state with the desired state (Dobies & Wood, 2020). This loop is shown in Figure 3, where the user's desired custom resource state is stored, and the operator automates the rest. This provides a very easy developer user experience where one defined custom resource is needed to deploy the required resource on the cloud. As one of the primary features of Kubernetes, using an Operator to orchestrate resources for Invoker is a major requirement for making the architecture completely cloud native.

## 2.3 Operators for Existing Engines

### 2.3.1 Flink Kubernetes Operator

As one of the most popular open-source stream processing engines, one of Flink's main roadmap to becoming cloud native is through the Flink Kubernetes Operator. (*Flink Kubernetes Operator Github*, 2023) The Flink Kubernetes Operator (FKO) is a open-source Operator written in Java using the Java Operator SDK (JOSDK) for running Apache Flink

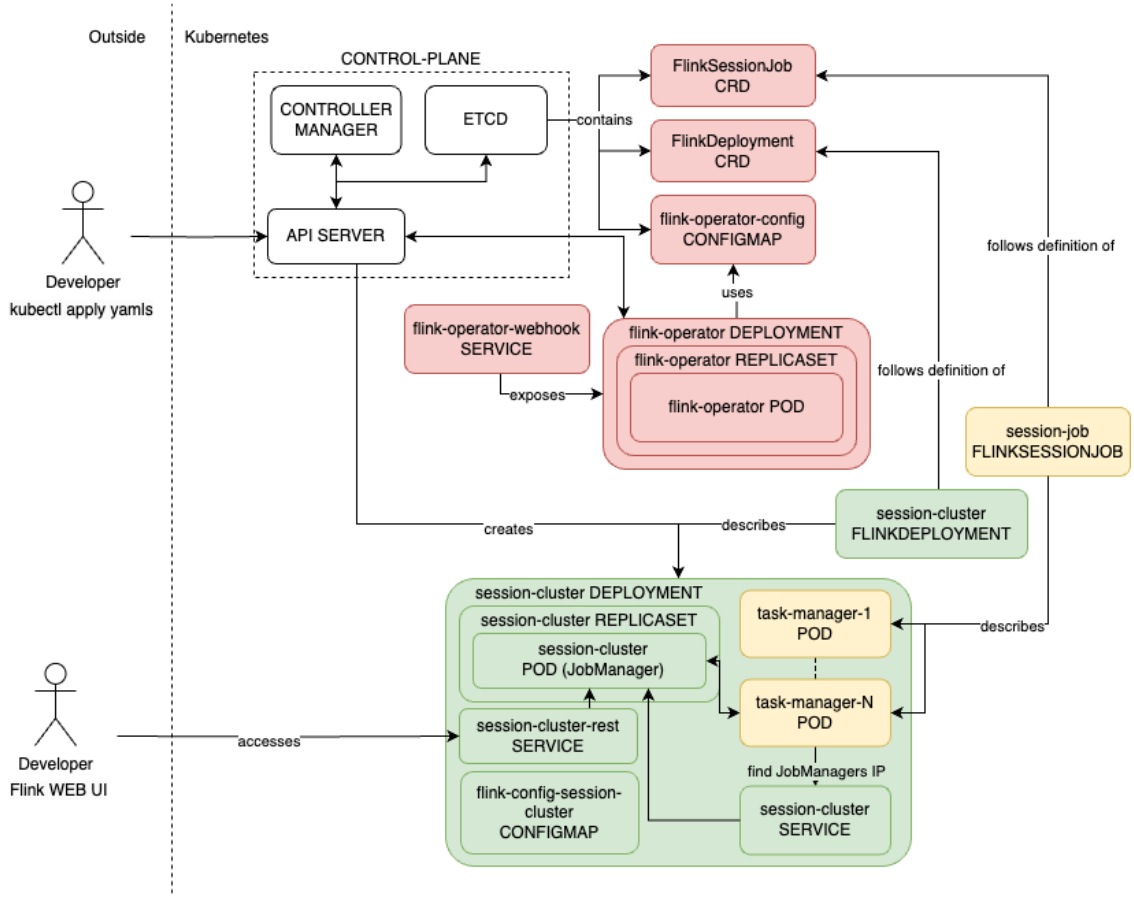


Figure 4: Flink Kubernetes Operator Resources on K8S

clusters on Kubernetes. Its Github repository is fairly new compared to Apache Flink, but within the two years of its lifespan, it has gotten a lot of attention and commits from the open-source Flink community. The basis of the FKO consists of two resources, FlinkDeployment and FlinkSessionJob. FlinkDeployment represents the whole Flink cluster. In Session mode, it is responsible for deploying the JobManager, but in Application mode, it would also support the TaskManager deployment. On the other hand, FlinkSessionJob represents individual jobs when running a Flink cluster in session mode. The overview of K8s resources a FKO deployment with a Session mode Flink cluster with FlinkDeployment and FlinkSessionJob

The Operator itself is deployed as a K8s Deployment on the cluster with the control logic within each Pod. Running a full FlinkDeployment cluster also involves a wide variety of Kubernetes resources deployed. First, is the Kubernetes Deployment, which, unlike traditional Deployments, owns the various resources depicted in Fig. 4. The primary units of computations, JobManager and TaskManager, are packages as Pods, with JobManagers running in a ReplicaSet, while TaskManager runs as individual Pods. As with all Kubernetes Operators, the primary way of submitting jobs and configuring the cluster

is to apply YAML files of the two mentioned Custom Resources, but as an engine, Flink also provides a web UI to access the Flink cluster. FKO allows this by using a Kubernetes Service, which exposes the Flink JobManager's IP addresses to a REST endpoint that Developers can access. Furthermore, to facilitate communication between TaskManagers and the JobManager within the cluster, Flink uses the session-cluster Headless Service. In Kubernetes, instead of exposing a set of Pods to one IP address and performing load balancing, it exposes all the IPs of the set of Pods, allowing them to be discovered using DNS, which facilitates network level Inter-Pod communication in Flink. Lastly, for Flink's cluster configuration, FKO uses a ConfigMap, which stores the configurations in the consistent Kubernetes native Key-Value (KV) store, Etcd. This ConfigMap is then mounted to the Pods as a Volume, which allows containers in the Pods to read from it as if it were a file in its directory. Kubernetes also will automatically update this mount, so as long as there is a reloading configuration logic in place, the configuration will always be updated.

The Flink Kubernetes Operator has also embarked on initiatives in dynamic reconfiguration and autoscaling on the cloud. In Kubernetes, the most basic way of scaling Pods is to use the K8s native Horizontal Pod Autoscaler (HPA). The way the HPA works is that it uses Kubernetes metrics collected by the Kubelet on each node and a desired metric provided by the developer to calculate the replicas needed to reach the desired metric. The formula used is below (Kubernetes, 2024):

$$desiredReplicas = ceil[currentReplicas * (currentMetricValue/desiredMetricValue)]$$

The HPA will then apply this new configuration to the existing Deployment, thereby scaling it up or down.

The Flink Kubernetes Operator has integrated support for the Horizontal Pod Autoscaler, allowing the scaling of TaskManager Pods to increase the capabilities of concurrently processing multiple jobs. However, Flink's inherent approach to parallelism diverges significantly. Within Flink, parallelism is managed via internal Task slots in TaskManagers, where each task slot signifies a unit of parallel execution capability. In a Flink Job's topology, every operation can execute in multiple parallel instances, enabling the same operations to process distinct data partitions. The quantity of these parallel instances is adjustable, allowing for optimization based on the specific demands of the job and the capacity of the available hardware resources. However, since the degree of paral-

lelism attainable is dependent on the total number of task slots available within the Flink cluster, this configuration is on the application level rather than the resource orchestration level.

To solve this issue, FKO built a native autoscaler for Kubernetes as a component in the Operator to make scaling decisions based on metrics Flink collects. (Michels, 2022) When a scaling decision is made, the FKO needs to reconfigure TaskManagers without restarting the Pods. Flink achieves this by only using the ConfigMaps for FKO configurations and Flink cluster level configurations, while scaling decisions made for TaskManagers are communicated to the Flink JobManager via the REST API service, leveraging the already existing endpoint for reconfiguring the parallelism for a specific job. Although Flink Kubernetes Operator's lifespan is much shorter compared to Flink, the immense size of the community has enabled it to contain many mature features that are valuable lessons in how to design and implement a Kubernetes Operator.

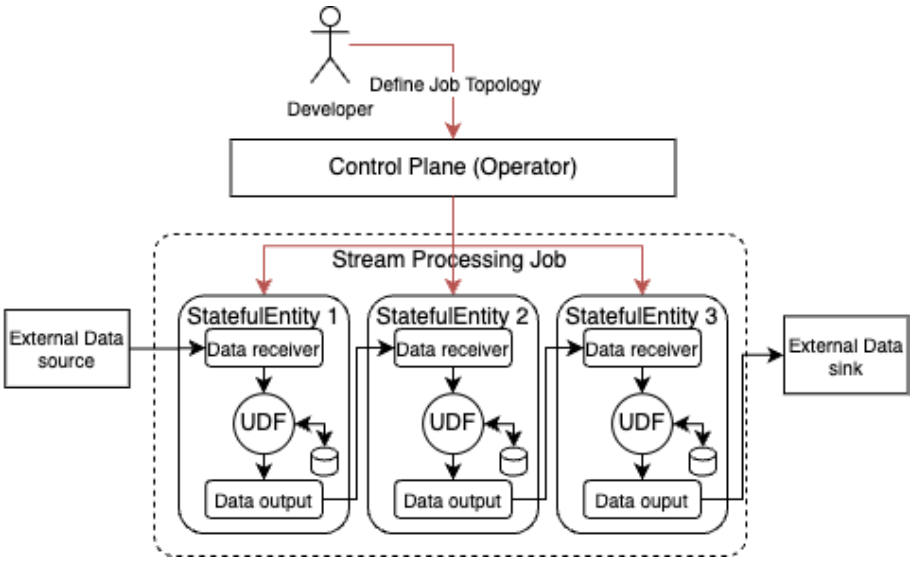


Figure 5: Invoker Programming Model Diagram

# 3 System Design

## 3.1 Programming Model

Before getting on to the Operator, the design of our SPE will heavily influence how the Operator functions. For our whole system, the topology will be made up of multiple Pods, communicating with each other through a messaging system. Fig. 5 shows an abstracted view of the programming model, with each UDF or operation (filter, map, reduce, etc.) being defined as a Stateful Entity, with storage available for stateful processing. The developer, through the Operator, will have control over the data flow from the data source to the Job, the output flow to the data sink, as well as the data flow between each Stateful Entity forming a topology. This programming abstraction allows the developer to form whatever topology they would like, giving them maximum control over the processing job. The role of the Operator, then, is to allow the developer to control the topology of the streaming job, using custom resource instances to define the topology as well as changing it during runtime to achieve dynamic reconfiguration.

The ideal workflow of a developer trying to create a stream processing job using Invoker on an existing Kubernetes cluster setup would be as follows:

1. Define and Implement UDFs using a set of SDK or Libraries in their preferred programming language
2. Containerize this program into a Docker Image to be run on Kubernetes
3. Define an InvokerDeployment instance detailing the topology of the job

4. Apply the YAML file and let the Operator deploy desired Kubernetes resources.

## 3.2 Designing Kubernetes Pods

In Kubernetes, each unit of compute resource is a Pod, and it would follow that each unit of execution in Invoker is a Pod as well. Hence, a stateful entity may contain multiple Pods running the same logic in parallel. We call each of these units of execution an Executor. The core execution logic of an Executor in a stream processing engine for a stateful function is as follows:

1. Accept a unit of data from an input stream. This could be a byte representation of a message.
2. Read from the state storage. (Optional)
3. Using the input data and the state data to perform an operation defined by the user, a User Defined Function (UDF).
4. Store the new state to the state storage. (Optional)
5. Output data that is needed to send to the output stream.

This is equivalent to an instance of Flink's stateful operation and Kafka Stream's singular node on its task topology. Making a Pod the basic unit of execution ensures that each Pod is much more lightweight than if it contained the entire topology and allows the whole system to be better suited to take advantage of Kubernetes' infrastructure in a cloud native manner.

## 3.3 External Communication and Storage System

Disaggregated systems in cloud native design is a fundamental principle that provides numerous benefits, aligning with the goals of scalability, resilience, and agility in cloud environments. Scalability is improved through the ability to manage each system individually and efficiently allocate resources for each system. Furthermore, in a disaggregated system, failures in one component are less likely to cause cascading failures throughout the application. This isolation helps in maintaining the overall system's availability and reliability. As a prototype, it also comes with the benefit of flexibility in choosing a technology stack for a system. Unlike traditional systems where the service is optimized for

specific hardware, in cloud computing, changing technology stacks can reduce costs and improve efficiency in different cloud environments.

In our design, we separated the communication system and storage from the units of execution. Instead of relying on individual Pods to communicate with each other, we decided to rely on an external messaging system. Similarly for state storage, relying on a separated storage plane means that each Pod is stateless, enabling faster and more fluid scaling. Since stateless Pods do not depend on the previous state to process requests, any Pod can handle any request at any time, making the system more resilient to failures as a whole. If a Pod crashes or is otherwise unavailable, other Pods can seamlessly take over without losing information or requiring complex recovery processes. This is also similar in principle to the various Function-as-a-Service (FaaS) platforms we investigated in the early stages of the project.

To enable communication between each stateful entity we chose Apache Kafka to enable scalable, at-least-once communication between multiple producers and multiple consumers. Kafka is a very mature system that has extensive development in the cloud computing landscape, including the business-oriented Confluent Cloud and the open-source Kafka Kubernetes Operator Strimzi. Kafka uses a Pub/Sub model, which allows multiple message producers to push messages into a topic that can be read by multiple consumers. This model of communication is mainly used for businesses with multiple microservices in a data pipeline but is also suitable for our scenario, where there are many parallel upstream and downstream Pods performing a one-way communication model. Similar to Kafka Streams, Invoker's design also leverages the Kafka topic partition feature to enable parallel communication and data processing. Each channel of communication between the upstream stateful entity and the downstream stateful entity can be defined as a topic in Kafka. By assigning each Pod in a downstream stateful entity to one or more partitions in the input topic, we can achieve flexible parallel processing with an ordering guarantee within each partition. As this Kafka cluster will be external to the set of stateful entities and Pods, it can be either deployed on Kubernetes using Confluent with an operator or outside the Kubernetes plane on bare metal machine. As long as the IPs of its brokers are able to be reached from inside the Kubernetes plane via networking policies, the Pods will be able to use it. Although setting up Kafka clusters and network communication can be put into the Operator, we decided this was outside the scope currently and manually setting up the messaging system. For my testing purposes, I used a Zookeeper Kafka cluster with a single broker deployed onto my local computer and accessed it through a

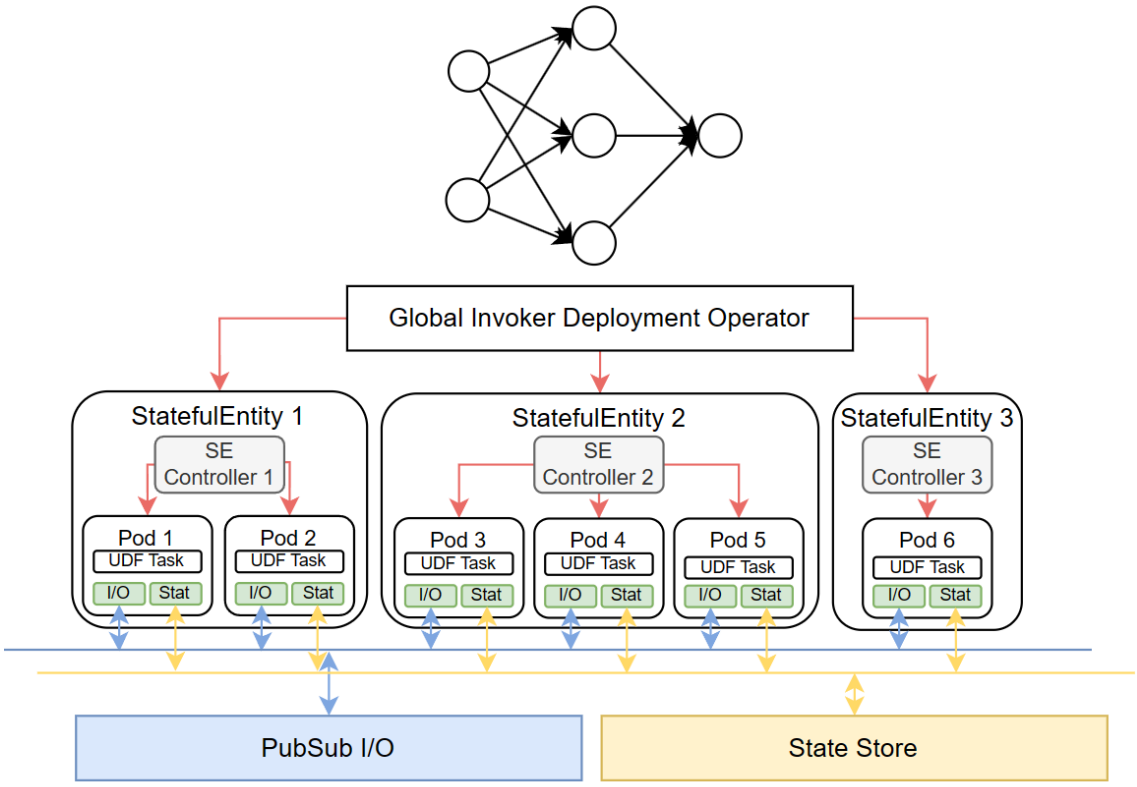


Figure 6: Invoker Distribution Model Diagram

local Kubernetes cluster.

As for the external storage system, the system is required to be a KV storage that is responsive, resilient, and scalable, preferably in a cloud native manner. This is an area in which further investigation can be done to design and select a suitable option. Existing options can be Redis, Kubernetes' internal Etcd, as well as other distributed KV storage like Zookeeper.

## 3.4 Distribution Model Design

A Kubernetes Pod will contain at least one container as its runtime, but decoupling the interactions with the I/O system and state system out from the main runtime will provide massive benefits. First, the decoupling allows for a better development experience for the developers writing the engine as well as the streaming job developers trying to create a topology. By separating state management from the core business logic and I/O operations, the system becomes more modular. This separation of concerns makes it easier to understand, develop, and maintain each component independently, as changes in one aspect (e.g., the storage layer) do not directly impact the business logic or I/O processing. With a decoupled architecture, it's also easier to replace or upgrade components without

affecting the rest of the system. For example, a developer could switch to a different database or storage solution with minimal impact on the business logic and I/O operations. As a stream processing engine, it is also able to fulfill the requirements of fault tolerance and resiliency. By distributing the point of failure, even if one container faces problems internally or with the external system, other components can run normally and continue to serve their purpose.

Combining the previously discussed programming model consisting of stateful entities and the external supporting systems, we can take a look at the distribution of Invoker on Kubernetes as shown in Fig. 6. The overarching Kubernetes Operator will manage the overview of the cluster, with stateful entities each managing a specific task. The SE controller in the image is a logical grouping, which will allow the Operator to manage individual stateful entities as separate systems. As mentioned before, each stateful entity will have one or more Pods, forming a 2-3-1 as an example in the diagram. Each Pod will consist of three containers running in parallel: The UDF Task running execution logic defined by the developer, an I/O sidecar container communicating with the PubSub I/O system (in our case Kafka), and a State sidecar container communicating with an external state store. The implementation of this system design comes in the implementation of these three containers and the Operator control plane which I will discuss in the next section.

# 4 Implementation

## 4.1 Implementing Invoker Pods

### 4.1.1 Sidecar Containers

Prior to implementing the Operator, I first had to have a working Executor Pod that could be run as a unit of execution. Using the design, I implemented the decoupling of components using the feature of sidecar containers in a Pod. Since Kubernetes allows for running multiple containers on a single Pod, having each component run in parallel containers running in the background allows for decoupling within each Pod while maintaining the co-location of each component on each Kubernetes node (as a Pod can only exist on one node at a time).

Currently, the sidecars consist of interfaces for the external systems, with the I/O sidecar implementing a Kafka consumer and producer. Although we want to eventually separate the storage plane, I currently implemented a local state storage for my prototype. One of the benefits of modularization is that if the storage system changes, the only change needed would be in the storage sidecar container as long as the inter-container communication system is kept the same.

One of the main implementation decisions in the Pod is the inter-container communication (ICC) which defines the protocol of communication between the Executor main container and its sidecars. Containers in a Kubernetes Pod share a network space, so it is very natural to have them communicate via localhost TCP socket or HTTP APIs. However, TCP sockets are in nature a very coupled blocking communication protocol, where the client will have to wait for each request and response, while HTTP is a protocol in the application layer, increasing the complexity of each request. Kubernetes also offers a feature called shared volumes, where Kubernetes creates a volume that can be mounted onto containers in a Pod. From the container's perspective, it is a shared directory that containers can read and write files from. Reading and writing from shared is a decoupled manner of communicating but is less efficient compared to network communication. The option we chose was a third option of using Unix Domain Sockets (UDS). UDS is a feature of the Unix operating system used for inter-process communication (IPC) on the same host. They facilitate data exchange between processes via paths in the filesystem, acting similarly to TCP/IP sockets but using the local file system as the address space. This means that processes can communicate with each other through a file on the filesys-

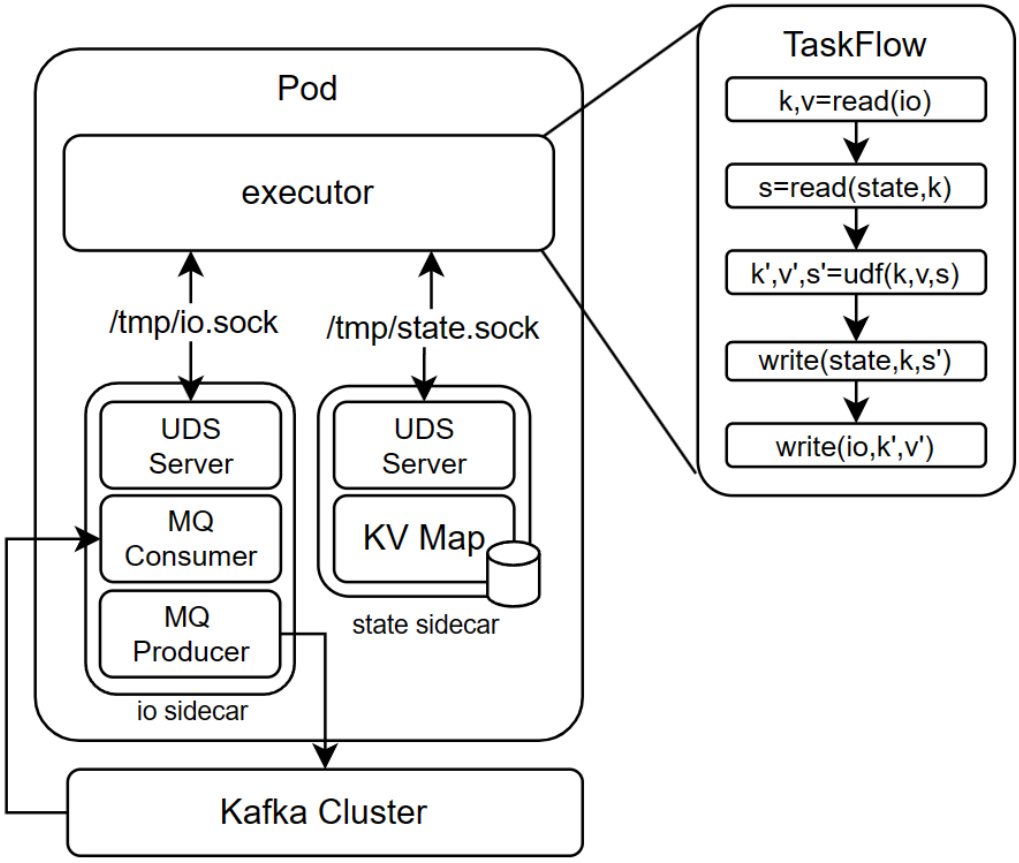


Figure 7: Container interactions within a Pod

tem rather than over network interfaces as long as they share the same IPC namespace. In combination with the shared volume provided by Kubernetes, UDS provides an efficient and high-performance method for IPC by avoiding the network stack overhead, making them an ideal choice for communication between containers on the same host. Furthermore, it is non-blocking, making it a better choice for the decoupled design of sidecar containers.

Fig. 7 shows the three containers in a Pod, as well as the inner logic of the Executor Pod. The UDS sockets are created in a temporary directory with servers hosted by each sidecar. In the I/O sidecar, the Kafka consumer polls from the cluster in a continuous loop, reading any new data for the input topic and immediately sending it to the Executor. It also forwards any send requests from the Executor to the output topic using a Kafka Producer. The state sidecar also only handles a read and write data from the UDS which reads and writes KV values to a local HashMap. Each container was compiled using Maven and through a basic Dockerfile that loads the JDK and jar file and runs it. Then they are uploaded to a public Dockerhub repository so that any Kubernetes cluster can retrieve and load it.

The language I chose to to implement these containers was Java using around 3000 lines of code. [1] This is mainly due to the familiarity and knowledge about Java and its surrounding libraries as well as learning from examples in Flink and Kafka Streams. The main library for Kafka APIs is also written in Java, with a lot of examples and documentation to learn from. Since the program will be containerized as binary compilations, the internal language isn't crucial to its functionality, and if another language is required, the modularization of the components allows them to be easily switched out.

### 4.1.2 Dynamic Configuration

The ability to be dynamically configured natively is one of the main motivations behind a cloud native SPE. Although a Pod is quite immutable under the Kubernetes context, application level configuration for each container can be changed. For our prototype, I identified a few essential configuration fields to have. These include the Kafka broker address, input topic name, output topic name, and input partitions. The configuration of the Kafka broker IP address and input and output topics plays an important role in allowing developers to create a versatile node topology. Although Kafka provides a robust mechanism for consumer registration and coordination through consumer groups, our approach involves manually assigning partitions to consumers. This low-level partition assignment enhances our control plane's capability for load balancing, leveraging partition-level metrics for optimized distribution. Such a strategy significantly increases the flexibility of our control plane. However, it's crucial to acknowledge that this level of manual partition management requires careful oversight. Proper management of these partition assignments is needed to avoid potential issues and ensure the system's overall efficiency and reliability. The above-mentioned 4 configurations are not the only ones that are possible, but for the prototype, I implemented these to prove how such a process would work.

Dynamic configuration management is achieved by integrating Kubernetes ConfigMaps with containerized applications. This approach involves creating a ConfigMap in Kubernetes and mounting it as a file within the appropriate containers, placed in a specified directory. Kubernetes automatically handles the maintenance of this mount, ensuring that any updates to the ConfigMap are promptly reflected in the files accessible within the containers. To exploit this functionality for dynamic configuration, a file watcher thread is implemented within the containers. This thread continuously monitors the ConfigMap

[1] https://github.com/yin72257/invoker-prototype

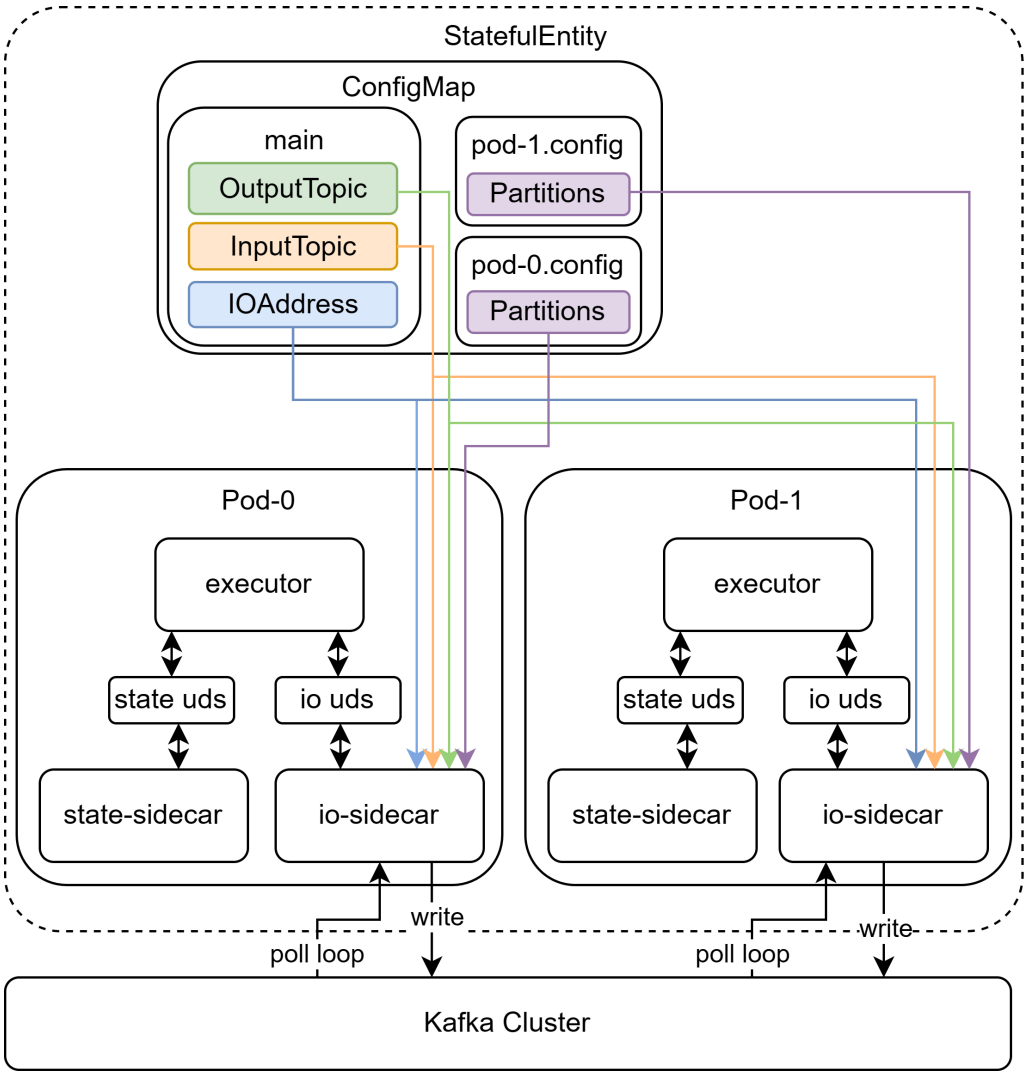


Figure 8: Kubernetes Resources for a Stateful Entity

file for any changes. Upon detecting a modification, it signals the main thread of the application. The main thread, in response, initiates a process to reload its configuration parameters based on the updated ConfigMap content. This mechanism ensures that the application can adapt to configuration changes in real time without the need for a restart, thereby enhancing its flexibility and responsiveness to evolving operational requirements. In conclusion, the Kubernetes layout of a stateful entity can be seen in Fig. 8 with the corresponding configurations color-coded. The application-level dynamic configurations are stored inside a common ConfigMap for a stateful entity, with one main file storing shared configurations while Pod-specific configurations are stored as individual files. Since the ConfigMap is mounted onto all the Pods in the stateful entity, a Pod will watch the main configuration file and the individual configuration files denoted by having the same name as the Pod. Without using a Kubernetes Operator, we can already create a stream processing topology by just writing ConfigMaps and Pods, with one example of this Pod

definition in Appendix A2. However, the benefit of Kubernetes Operator is that it simplifies the deployment process significantly, which I will describe the implementation process in the next section.

## 4.2 Implementing Kubernetes Operator

As mentioned before, the Operator's role is to transform the custom resource YAML created by the developer into resources on Kubernetes which forms the topology of Pods. There are a lot of resources discussing the best practices in writing Kubernetes Operators made by different organizations such as Red Hat and Cloud Native Computing Foundation (CNCF). Originally starting with Etcd Operator and Prometheus Operator (an open-source monitoring application) in 2016 (Phillips, 2016), the Kubernetes operator landscape has evolved drastically since with many open-source projects hosted on GitHub as well as CNCF's OperatorHub.io (*OperatorHub.io*, 2023). To interact with Kubernetes built-in resources, APIs need to be called, whether through command line commands using Kubectl or Go packages (*client-go*, 2023). The client-go package provided by Kubernetes is the default way of programming Go controllers. This package provides ways to use custom SharedInformers to list and watch custom resources on the etcd. SharedInformers are essential for the logic of a Kubernetes Operator, and they work as follows. When putting a watch on its assigned resource, and a change is detected, it will use a delta FIFO queue to push the changes to its internal cache controller and indexer components. The developer's role, then, is to program the logic of the Custom Controller, writing handler functions as listeners to respond to changes in the Informer. By customizing the handler functions with other client-go APIs, mainly CRUD functions for native Kubernetes resources, the controller's role is to ensure that the current state of the Kubernetes cluster is the same as the described state of the custom resource instance. (*Kubernetes Informer pkg*, 2024)

Kubernetes' Special Interest Groups (SIGs) have also developed wrapper libraries designed to allow for a smoother operator development workflow, abstracting away logic that may need to be repeated frequently when writing a controller with client-go. This abstraction comes in the form of the open-source controller-runtime library with over 2700 commits. This controller-runtime library removes some of the ability to do some of the fine-grained customization, but it neatly packages the common read, update, and change operations to a reconciliation loop. The Operator Framework toolkit, part of the

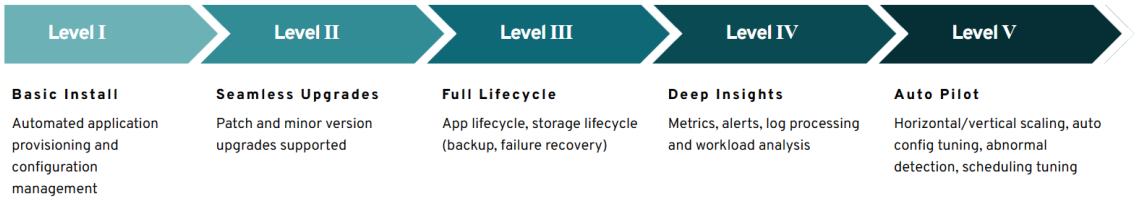


Figure 9: Operator Lifecycle (*Operator capabilities*, 2024)

CNCF, is a set of tools for developers used to build and deploy operators on Kubernetes (*Operator Framework*, 2023). The OperatorSDK can be used to bootstrap and scaffold the necessary Go code for implementing the watch and modifying the control loop. It also provides easy to use commands to develop and test the operator as it is being implemented through the use of a Makefile. I chose to use Operator SDK in Go to implement my operator in Go. Since Kubernetes is written in Go, and the default language of Operator SDK is also Go, choosing Go as the language to write a cloud native application's resource orchestration logic is an obvious choice. The alternative was to write the controller in Java using the JOSDK, using the same language the Executor container and sidecars were written as to match the languages. However, I did not choose this option as writing the operator in Go makes it so that the resource orchestration layer is decoupled with the execution layer, and that in the future if needed, the executor can be adapted to Go as well.

Operator Framework defines an operator's level of maturity using a model of five capability levels. The basic description is shown in Fig. 9. This model can be used as a way of programming a fully mature Kubernetes Operator from the bottom up. Each level can be fulfilled through several criteria.

1. Level 1: Basic install. The first level pertains to the basic ability to be able to run the Kubernetes resources by applying a defined custom resource. Any operator fulfills this level by having its most basic function of provisioning cloud resources.

2. Level 2: Seamless upgrade. This level is fulfilled by having the ability to upgrade the resources running on Kubernetes by updating the custom resource while maintaining the lowest level of disturbance.

3. Level 3: Full Lifecycle. Here, the operator should be able to manage the lifecycle of the custom resource completely, including deleting and restoring failed versions. An Operator is fully functional at Level 3, while Levels 4 and 5 are extra functionality.

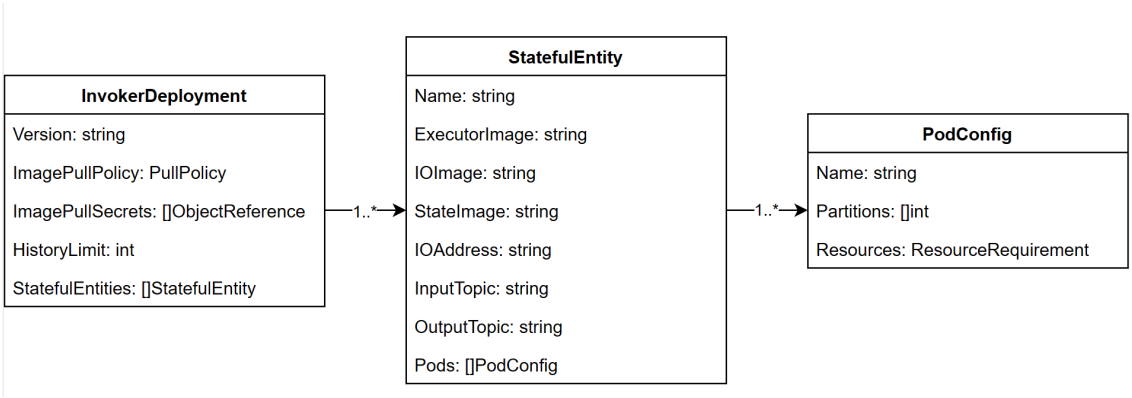


Figure 10: InvokerDeployment Custom Resource Definition UML

4. Level 4: Deep Insights. An operator reaches this level by integrating a metrics or logging system. This can either be done with custom metrics and frameworks, but more commonly it is done with a Prometheus cluster. Prometheus is an open-source monitoring system that has made a lot of improvements to fit very well in the cloud native ecosystem.

5. Level 5: Auto Pilot. The final level requires the use of previous metrics to conduct autoscaling on the resources managed. Kubernetes natively has HorizontalPodAutoscaler (HPA) and VerticalPodAutoscaler (VPA) that can be integrated within the operator for general Pod scaling purposes, but many operators such as Flink Kubernetes Operator and KubeRay will choose to have a more custom autoscaling logic that takes into account the nature of the app.

   The previously discussed Operators mostly fit into the Level 5 maturity level. For this prototype project, I will be reaching Level 4 in terms of maturity, with autoscaling being a feature we investigated, but have not decided on a design yet.

These levels are cumulative and subsequent levels can only be reached when the previous level is reached. For instance, Level 5: Auto Pilot requires insights from metrics and logging from Level 4: Deep Insights to decide when to scale a resource out and in.

### 4.2.1 CRD Design

The definition of resources in Kubernetes can be split into two parts: Spec and Status. The Spec or specification denotes the fields filled in by the developer, which defines the state of the custom resource on the Kubernetes cluster. The Status are fields automatically generated by controllers of the resource which denotes the status of that resource currently in the cluster. In most cases, this status marks whether a resource is ready, upgrading,

deleting, etc. The design of the spec took many iterations to decide on the best way of representing the configuration of an Invoker cluster. Fig. 10 displays this custom resource definition Spec in UML format.

The custom resource definition's design is based on the model of the processing framework. One instance of an application built using the Invoker engine is called an InvokerDeployment. A Kubernetes cluster can have multiple InvokerDeployments, with each representing a stream processing job with a topology of UDFs. Figure **??** shows a filled in YAML of FlinkDeployment with all its configuration fields. The configuration fields can be split into 3 layers from the largest to the smallest.

1. InvokerDeployment: Configuration meant for the entire application.

2. StatefulEntity: Each Stateful Entity in the InvokerDeployment has its own level of configuration. Fields at this level will be applied to every Pod in the Stateful Entity.

3. Pod: Each Pod in a stateful entity will have its own configuration which differentiates the behavior of a Pod in a stateful entity from one another.

This configuration layer on top of each other, in the sense that a Pod will be able to use a configuration set on the InvokerDeployment level. The levels are more of a logical concept to determine which configurations are common for resources at a certain level. Each configuration field in the custom resource Spec can also be categorized into two types: Kubernetes configuration and Application Configuration. Kubernetes configuration refers to fields used by the operator to directly generate Kubernetes resources. For example, the image of the Pod the operator is running can be categorized as a Kubernetes configuration. Application configuration, on the other hand, refers to the configuration used by the application runtime. For example, determining which Kafka topic to connect for the input sidecar to is an Application configuration. These are placed inside a ConfigMap to allow the Pods to read from and parse when initializing. Changing them would also allow for dynamic configuration without needing to restart the Pod.

InvokerDeployment Level Configurations

Currently, the InvokerDeployment level configurations are all Kubernetes configurations.

- Version (String): A common field used by most native and custom Kubernetes resources to determine the version of InvokerDeployment used.

- ImagePullPolicy (String): A native Kubernetes resource defining when Pods will pull their container images from DockerHub or load them from the local registry
- ImagePullSecrets (List of LocalObjectReference): An optional list of object references to secrets, for cases where the DockerHub Image is in a private repository and a secret is needed to access it.
- HistoryLimit (Integer): Number of ControllerRevisions the operator will keep in the etcd for an InvokerDeployment resource.
- StatefulEntities (List of StatefulEntity Configurations): Next level of configuration in a list format.

Stateful Entity Level Configuration

Kubernetes Configurations

- Name (String): Name of the StatefulSet used to manage the Pods of a Stateful Entity
- ExecutorImage (String): Image of the core executor task. Defined and compiled by the developer and is a required field. The Docker container image will contain the running logic of theread, process, and write of the core UDF.
- InputImage (String): Image of the input sidecar container image. This image can be provided by the framework, and the developer won't have to modify it. This also applies to the StateImage and OutputImage.
- StateImage (String): Image of the state sidecar container image.
- OutputImage (String): Image of the output sidecar container image.

Application Configurations

- IOAddress (String): Configures the IP Address the input sidecar and output sidecar will interact the messaging system with. In the case of using a Kafka cluster, it is the broker IP address.
- InputTopic (String): Configures the input Kafka topic for the input sidecar. If we want the input sidecar to become more generic to accommodate other types of messaging systems, a different name might be needed.

```yaml
resource:
  claims:
  - name: cpu
  - name: memory
  limits:
    cpu: 1
    memory: "1Gi"
  requests:
    cpu: 500m
    memory: "512Mi"
```

Figure 11: Example of Kubernetes ResourceRequirement Limiting the lower and upper limit of CPU and memory to between 0.5 core to 1 core and 512 Mi to 1 Gi respectively

- OutputTopic (String): Configures the output Kafka topic for the output sidecar. Similarly, it might be renamed if another messaging system is implemented.

- Pods (List of Pod Configs): List of the configuration of Pods. The length of the list controls the number of parallel instances of Pods.

Pod Level Configuration

The current Pod level configuration is only the partition list and resource requirements as the current setup is sufficient for a working prototype.

- Partitions (List of Integers): Application Configuration. List of partitions used as input to manage input load from Kafka. This setup is similar to how Kafka Stream distributes the load from its input topic.

- ResourceRequirement (ResourceRequirement): Kubernetes Configuration. A native Kubernetes kind that is able to define the resource requirements of each Pod. A Kubernetes ResourceRequirement can be customized in many ways, the two main ways being a lower limit and an upper limit. As shown in 11, setting a claim and giving an upper limit and a lower bound of request ensures that Kubernetes will allocate the Pod enough resources.

Status Fields

Status is the second part of a Custom Resource Definition. They are not set as instance fields but are managed by the Operator to record the current status of the custom resource instance. Here are the ones I have chosen based on common practice:

- Components (ComponentStatus): An Object containing the Statuses of the ConfigMaps and StatefulEntities the Operator has to manage. To simplify this manage-

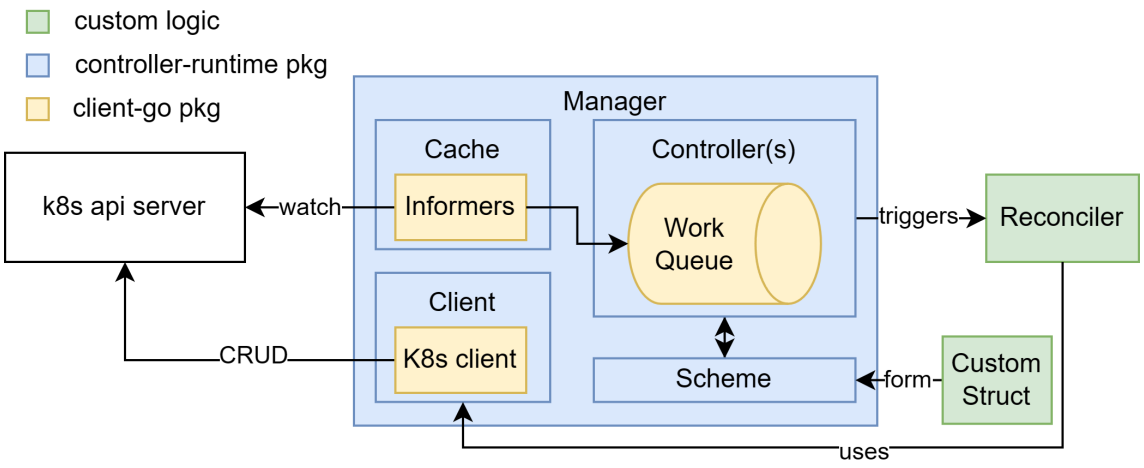


Figure 12: Architecture of important components of the controller-runtime package

ment, it only records whether every ConfigMap/StatefulEntity Pod is ready using an AND statement.

- CurrentRevision (String): ControllerRevision name of current Spec.
- NextRevision (String): ControllerRevision name of the next Spec. Used for when updating the custom resource instance.
- LastUpdateTime (String): String representation of the last the custom resource instance was updated.
- State (String): A general indication of the readiness of the state. If all stateful entities are running, the state will be Ready.

One such instance of this custom resource can be seen in Appendix A1 two stateful entities with one Pod each.

### 4.2.2 Go Operator with Controller Runtime

Controller runtime requires a unique pattern of implementing the reconciliation loop, and as a library, it has a complicated inner architecture to enable writing the Operator. Fig. 12 showcases the main components of this architecture from a high level. At the core of an Operator is the Manager, capable of housing one or several Controllers. Each Controller is responsible for managing a specific Custom Resource Definition (CRD). Additionally, the Manager employs a Cache to oversee the Informer logic from client-go, while the work queue for each custom resource is established by the Controller. To interact with the Kubernetes API server for CRUD operations, a Client component is necessary.

From the viewpoint of a developer, the initial step involves defining the custom resource, recognized in Kubernetes terminology as an API. This process is accomplished by creating a Go struct that includes both Spec and Status fields. This struct is subsequently transformed into a native Kubernetes object through the application of a Scheme provided by the controller-runtime package. Following this, the main reconciliation loop is encapsulated within a singular Reconciler function. This function is activated by the Controller in response to changes in the watched resources or when a reconciliation is intentionally invoked.

The controller-runtime package offers an invaluable abstraction layer, simplifying the intricate management of Informers, Indexers, and handlers found within the client-go package. This abstraction not only streamlines the development process but also enables the creation of a simplified yet fully operational Operator.

### 4.2.3 Operator Workflow

Other than the central reconciliation loop, there is not a standardized pattern of writing the code of a Go Operator. KubeRay has a flat code structure, where most of the controller logic for each custom resource is packaged into a few Go files, and the Postgres Operator attaches all the functions required to the main controller class. Since there was no set standard for structuring Operator code, I decided to follow closer to a logic used by the Flink Kubernetes Operator. As Flink's deployment of resources on Kubernetes is fairly complex, it separates the reconciliation loop into four phases observe, update, convert, and reconcile, each handled by a module in the code. This structure provides modularization of the code and logic, and if it is worked on further in the future, it improves readability and extensibility as long as the four phases are understood. The main functions of each phase are the following:

1. Observe: Get the resources the controller manages from the Kubernetes cluster using List and Get methods from the client-go library.

2. Update: Depending on the observed state, update the status of the managed Kubernetes resources. If an update is required, early exit and restart the reconcile loop.

3. Convert: Use the user defined custom resource to form the desired Kubernetes resources as Go objects.

4. Reconcile: Reconcile the current state of the Kubernetes cluster from the observe

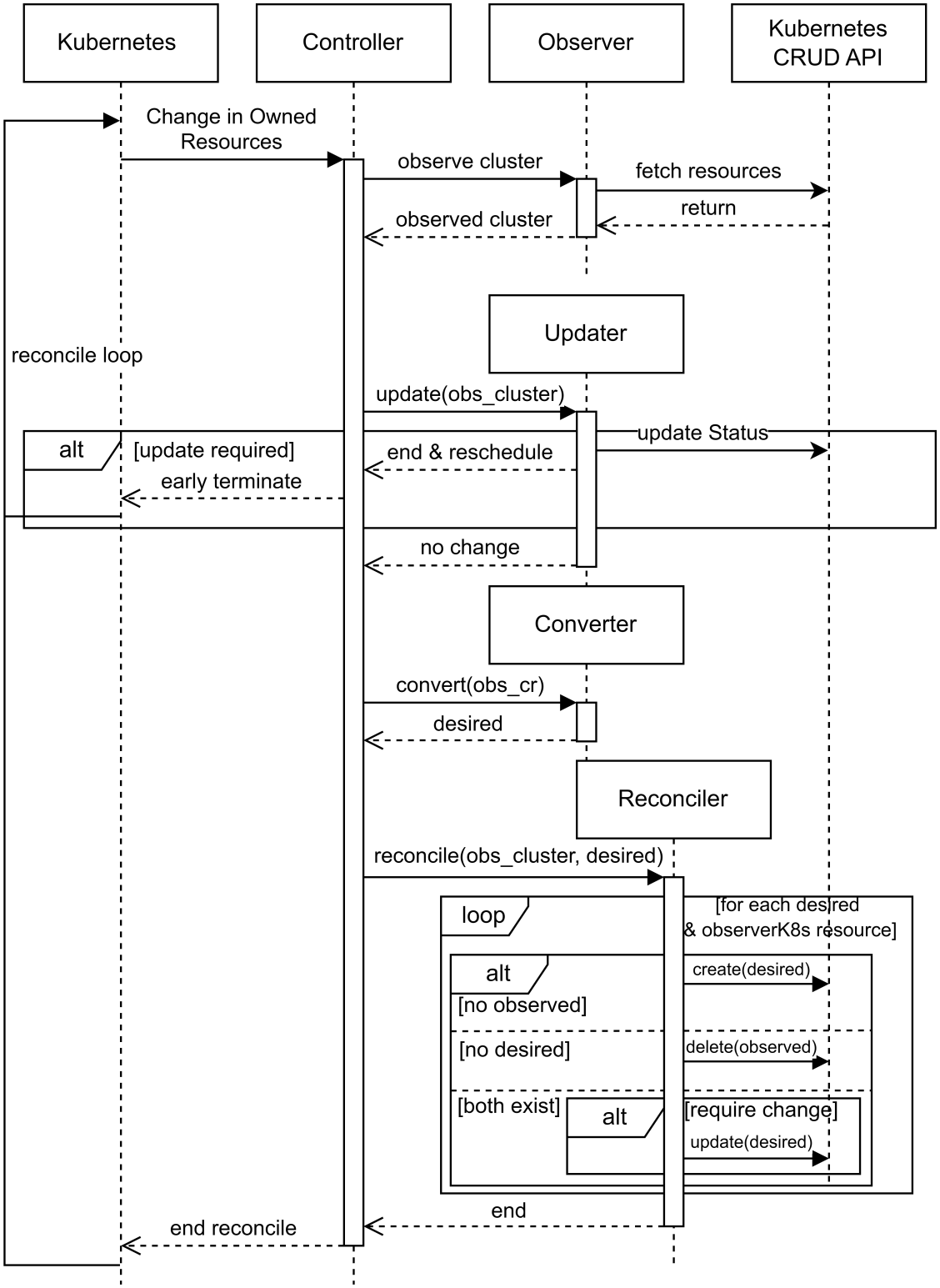


Figure 13: Sequence diagram of reconcile flow

step with the desired state from the convert step by either Creating, Updating, or Deleting selected resources in the Kubernetes cluster.

Fig. 13 showcases the full reconciliation loop of the Invoker Kubernetes Operator. By separating the responsibility of each component, it makes the code more readable and extensible which benefits debugging as well as potential collaborators.

### 4.2.4 Levels 1 and 2

By implementing the above workflow with the controller-runtime package, the Operator is able to complete Level 1 and Level 2. Writing the reconcile loop of a controller-runtime Operator requires a unique approach to programming, particularly because of the idempotent nature that must be maintained within each reconciliation iteration. The main challenge is that the reconcile function must be idempotent, meaning it can be called multiple times with the same inputs without causing unintended side effects. This requirement ensures that even if the reconcile loop is triggered repeatedly—due to external changes or events—the state of the system converges towards the desired state. The functionality at this level includes being able to define and apply a user defined custom resource YAML onto the Kubernetes cluster where the Operator will create the ConfigMaps and Pods with the correct configuration, as well as being able to update the basic fields of the custom resource such as version and name. Through the updater, the status fields were also able to be changed between not ready to ready.

### 4.2.5 Level 3

To achieve a fully functioning lifecycle, another important aspect is its dynamic configuration. Based on referencing existing operators, I used the ControllerRevision feature of Kubernetes to achieve both dynamic configuration of the managed resources and the ability to recover previous versions of the custom resource. It acts as a snapshot mechanism, recording the configuration or state of a controlled resource at a specific point in time. This allows for versioning of the resource's state, enabling features like rollback to a previous version in case of issues or undesirable changes. Each ControllerRevision contains a unique revision number, which increments as new revisions are created, and a data field that holds the serialized representation of the resource state. Normally, it is used for StatefulSets, but as a resource, I can implement it within my own custom operator for InvokerDeployments.

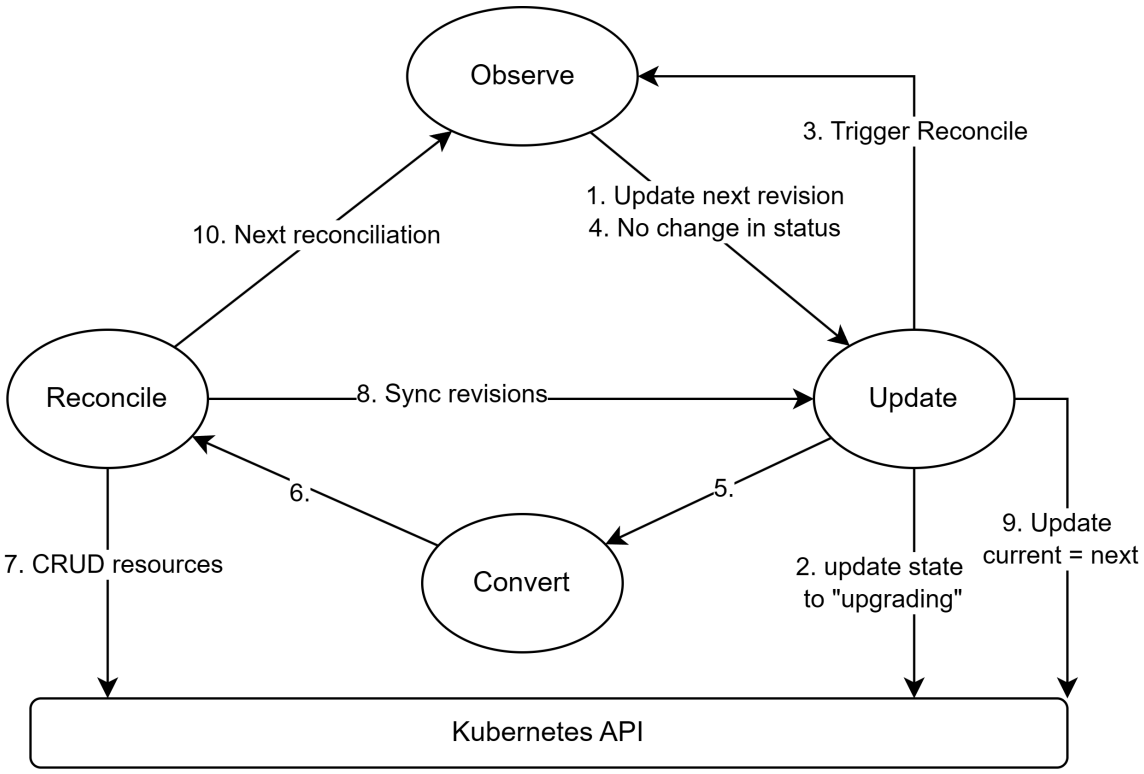


Figure 14: Updating Revision statuses

I implement this feature using the two status fields: CurrentRevision and NextRevision which work together when an InvokerDeployment instance is updated. CurrentRevision represents the revision name of the resources currently deployed on the Kubernetes cluster, while NextRevision represents the revision name of the new instance of InvokerDeployment. These revision names are calculated by performing a recursive hash on all the fields of the observed custom resource instance.

Fig. 14 showcases a flowchart of how updating an InvokerDeployment instance would be handled. First, the observer updates the next revision name, the updater will then realize that the resource has been updated and change the status fields of the InvokerDeployment on the Kubernetes cluster to Upgrading. This immediately triggers another observe to ensure the read status fields are up to date. Steps 5 and 6 are standard procedure, as well as step 7 which updates the Kubernetes cluster. After updating all the resources, the reconciler will sync the revisions, which triggers the updater to update the current and next revision status as well as create a new ControllerRevision instance that records the updated Spec in a string format. This step also deletes the earliest ControllerRevision if the total number passes the set HistoryLimit in the Specifications. Each unique InvokerDeployment Spec will only have one ControllerRevision instance. If a version is switched back, the revision will be updated with a revision number of the current revision

number + 1. Although this design won't allow developers to see every revision change in detail, it is efficient in viewing all past versions of a custom resource.

By correctly implementing status fields and ControllerRevision, I was able to allow updates to the instance to be reflected in the Kubernetes cluster in a dynamic manner, completing the requirements for a Full Lifecycle management Operator.

### 4.2.6 Level 4

Prometheus is an open-source monitoring and alerting toolkit widely used in cloud-native environments. It's designed for reliability and efficiency, allowing users to collect and process metrics from various sources. Prometheus enables the definition of highly flexible queries and alerts based on the time series data it collects. This data can be gathered via a pull model over HTTP, pushing through an intermediary gateway, or via service discovery in dynamic environments. For Kubernetes, open-source Prometheus Operator and Helm charts allow for easy and seamless integration with applications on the Kubernetes cluster, making it very suitable for monitoring the Invoker Operator.

The controller-runtime library also supports native support for Prometheus Operator by exposing its metrics through the /metrics endpoint using a ServiceMonitor custom resource instance. The default metrics include number of total reconciles, work queue latency, and others to form a general overview of the condition of the Operator. As for the operands, or the Pods that the operators are controlling, their resource usage metrics can be exposed by a cAdvisor component embedded into the Kubelet in each node. Although the Invoker Operator does not have auto-scaling functionality yet, being able to access Pod metrics through the cAdvisor exposer will be useful in determining when to scale.

To achieving a Level 4 Operator, I incorporated the kube-prometheus-stack into my Kubernetes cluster and Operator. (Prometheus, 2024) In the next section, I will be using metrics exposed to Prometheus to conduct my testing and analysis. Overall, my Invoker Operator acts as a fully functioning prototype and template in providing the resource orchestration plane for a stateful stream processing system on Kubernetes. Using the controller-runtime package and the OperatorSDK, the code took around 5000 lines to implement. [2]

[2] https://github.com/yin72257/go-invoker-operator/

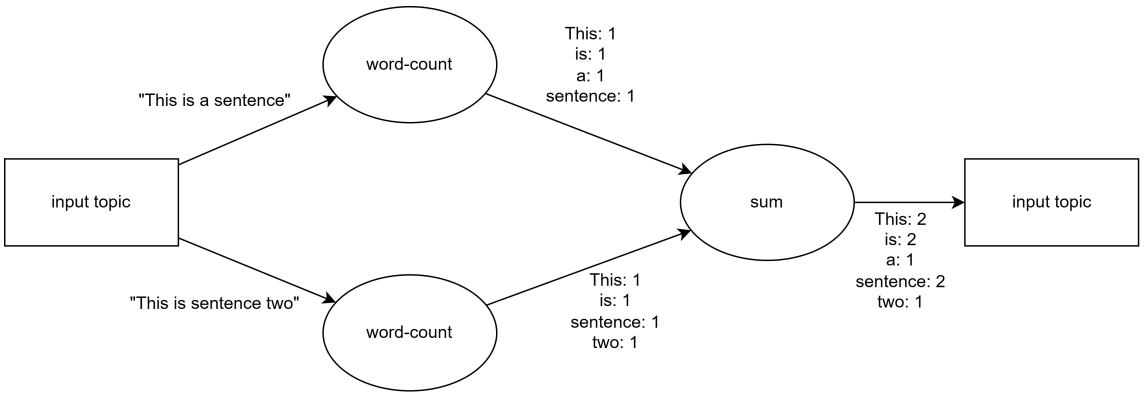


Figure 15: Updating Revision statuses

# 5 Evaluation

## 5.1 Sample Processing App: Word Count

Although Invoker is not at the stage of being able to be tested in comparison with existing Stream Processing Engines, we can still ensure its correctness and functionality through testing. To facilitate testing, I am using a sample word count application. I will be using two Stateful Entities with two operations. The first stateful entity will be multiple nodes of stateless operations where for an inputted string, the Executor will count the occurrences of each word. This is then passed as a key-pair value to a downstream task which sums up the occurrences, keeping track of the previous sum, and outputs it to a Kafka topic. This is a common application used to test stream computation that involves both stateful operations and parallel processing. Fig. 15 shows the operations of a word count application with a 2-1 topology. To set up this test, I am using a simple Java Kafka Producer with the default Kafka Partitioning scheme which assigns each message to a random partition if it contains no key. For my testing, I will be running the Kafka Producer in the background, producing a preset text in 5-word phrases repeatedly with one second in between each send. This will simulate a streaming data source for this application. The tests below were all run on a Minikube Kubernetes cluster with 8 CPUs and a 14GB memory limit on an Apple M1 processor. For this evaluation, the hardware doesn't particularly matter as the purpose is to verify functionality and correctness.

## 5.2 Scaling Components

Since parallelism is limited by the partitions of the input Kafka topic, to evaluate dynamic scaling capabilities, our test environment is configured with a 6-partition input topic. This

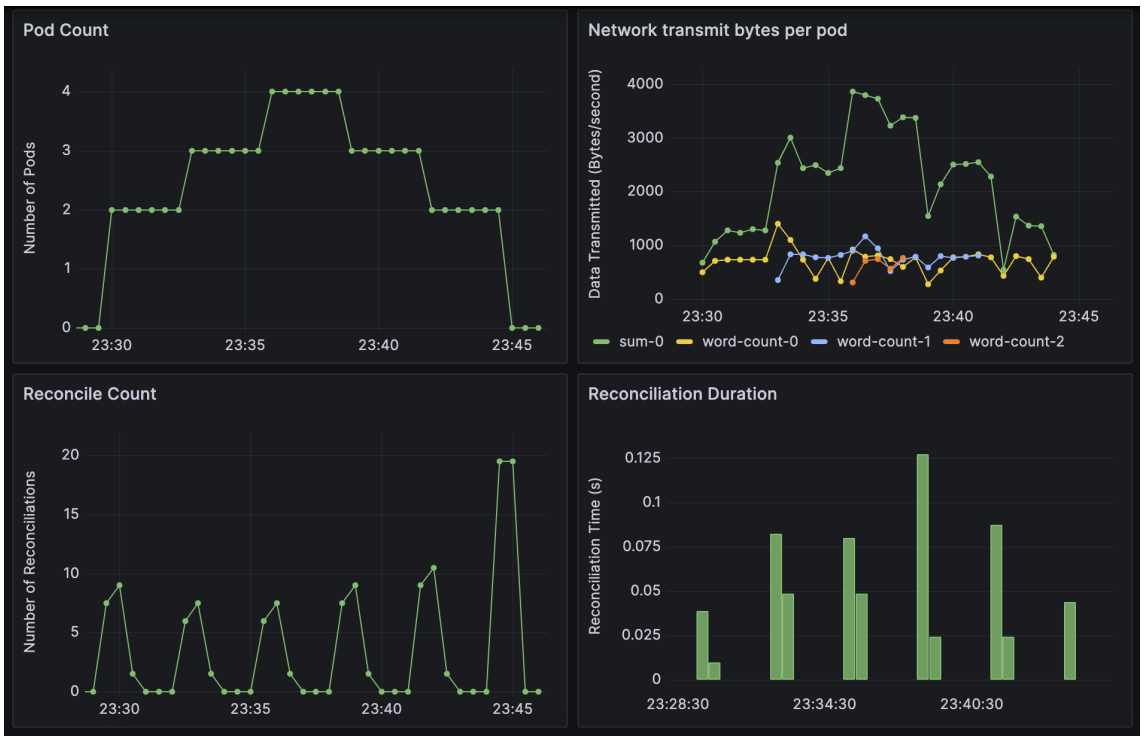


Figure 16: Scaling Component Metrics

setup facilitates an incremental adjustment of parallelism, increasing from 1 to 3, to assess system performance and resource allocation efficiency. At the initial parallelism level of 1, a single Pod is designated to process messages from all partitions, labeled "0,1,2,3,4,5". As we increase parallelism to 2, the distribution of partitions becomes segmented between two Pods; the first Pod manages partitions "0,1,2", while the second Pod handles "3,4,5". This pattern continues, further dividing partition responsibility as parallelism escalates every 3 minutes, ensuring an even split across available Pods. After hitting the maximum parallelism of 6, the cluster will slowly scale down to parallelism=1 in the same manner. This approach allows us to systematically observe the scalability and load distribution effectiveness of our system in a controlled manner. At the same time, I changed the frequency of the Kafka Producer's inputs to simulate a varying input load. This process of changing input load and reconfiguring was set up in a Java project with two parallel threads.

The initial metric we examine is the Pod Count. With a parallelism setting of 1, the configuration includes one word-count Pod alongside a single sum Pod. As the parallelism level increases to 4, the configuration expands to include three word-count Pods and maintains one sum Pod. Additionally, the network transmit metrics for each Pod offer insights into the input load handled. Given that all topology data flows through the sum-0 Pod, its network input/output (I/O) experiences an uptick with increased load.

The word-count Pods, on the other hand, distribute input partitions evenly, resulting in a consistent load across them throughout the testing phase.

As for the measurement of the Operator, the Reconcile Count is used to track the number of reconciliation loops executed within the controller-runtime code. This figure represents the controller's activities (creation, updating, deletion) as well as the status updates to ensure operational accuracy. The Reconciliation Duration metric reflects the time required for the controller-runtime to achieve a stable state for the custom resource, highlighting the efficiency of the reconciliation process.

## 5.3 Dynamic Reconfiguration

Another main feature of Invoker Operator is dynamic configuration. The primary method of changing application level configuration is through reassigning the partition assignment between Pods using the Custom Resource Spec. For reassigning input partitions, a Pod is able to dynamically configure without restarting, only resetting the consumer in the I/O container. For our test case, I use the same round robin producer for a word count application. By testing with a parallelism of 2, the first configuration is an even split of partitions with the first Pod being assigned "0,1,2" while the second Pod is assigned "3,4,5". After 6 minutes, the second configuration is applied where the first Pod is assigned "0,1,2,3,4" and the second Pod is assigned "5". This should result in a 5:1 split of load balance. The result of this test is seen below.

As seen from the network transmission in Fig. 17, the reconfiguration was successful and the load of data is able to be configured from the Operator. This reconfiguration also only occurs using container level logic, meaning the Pod will be running without any interruptions as evidenced by observing Kubernetes logs as well as the Pod count metrics being consistent. The Operator is able to achieve this in less than 10 milliseconds on average.

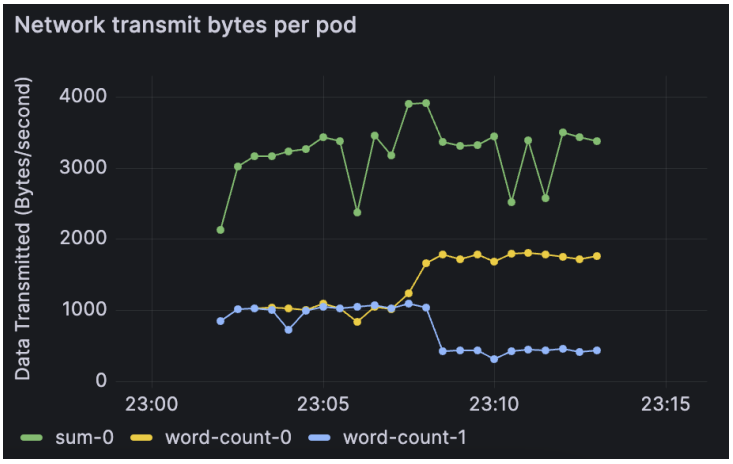


Figure 17: Network transmission after dynamic configuration

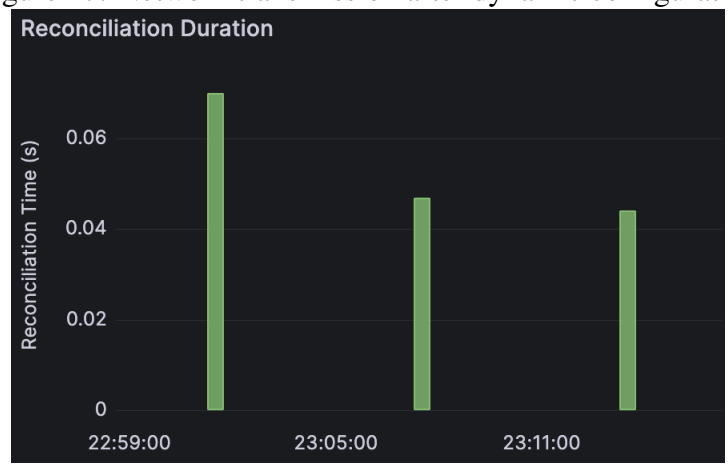


Figure 18: Duration of reconciliation in Dynamic Configuration


Figure 19: Number of Pods during reconciliation

# 6 Summary

## 6.1 Further Work

The present version of the Invoker Executor runtime and Invoker Operator lays the groundwork for future enhancements. To fully realize a cloud-native stream processing system capable of competing with established platforms such as Apache Flink, it is essential to design and implement additional functionalities that enhance performance, resilience, and feature set. However, we believe that these are achievable with tools provided by the cloud, and through further investigation and testing, we would be able to decide on the most optimal solutions to overcome these challenges.

For example, we can improve the external systems Invoker currently relies on. Choosing the right storage system would require further design of the state sidecar and investigation into the selected system. This selection process could include evaluating systems such as Redis, Zookeeper, Etcd, or others that align with our specific requirements. A comprehensive analysis of the advantages and limitations of each system, their reliability guarantees, and the unique challenges they might encounter within a cloud-based environment is vital for the development of an effective, robust stateful architecture. Another area of improvement is the selection of our message distribution system. Although Kafka is widely used in the industry and is highly scalable and performant, newer systems such as Apache Pulsar or a Service Mesh may be more in line with the cloud native paradigm. Similarly, extensive research can be done on these systems to determine the most suitable one and its configurations.

It is also essential to feature align Invoker with existing stream processing systems. This would require the Operator to be improved on as well. Although it fulfills the features of a Level 4 operator maturity level, effective autoscaling based on internal load metrics would require further design and implementation. Control plane level load balancing and autoscaling could either be integrated within the Operator or be implemented in a completely separated component, only using the Operator as a resource orchestration tool. The control plane could also be responsible for maintaining backups and failovers, using ControllerRevisions to roll back to previous configurations. App and Kubernetes configurations would also need to be added to the custom resource definition to allow developers to be able to customize their streaming job's needs.

Furthermore, the current way of deploying jobs using custom resources can be im-

proved upon as well. Instead of having to write long YAML files to define custom resources, it would be a better user experience to use a library to code their user defined function which can be interpreted into a YAML file through a system that also goes through a validation process. A UI component could also be integrated into the system, similar to Flink UI's REST API. These features may impact the types of resources managed by the Operator but are easily implementable using the existing modularized code.

## 6.2 Conclusion

The trajectory of stream processing engines has seen a remarkable evolution over the years, culminating in the emergence of Cloud Native as the inevitable next phase in this progression. Traditional industry solutions have enhanced mature engines with cloud integration tools and features, yet transitioning to a next-generation stream processing engine demands a radical redesign. This redesign must place Cloud Native principles at its core, fundamentally altering how stream processing is conceived and executed in a cloud-centric world.

In pursuit of this vision, my work with Invoker on Kubernetes represents a significant milestone—a fully functional prototype that embodies the principles of Cloud Native from the ground up. Drawing on the lessons learned from existing cloud integrations and leveraging the tools provided by the Kubernetes ecosystem, we designed and implemented the resource-level components that underpin Invoker. Our approach has been holistic, considering not just the technical requirements of a stream processing engine, but also the cloud native way of managing Kubernetes with Operators. As we look forward, the prototype of Invoker on Kubernetes is just the beginning. The insights gained from this endeavor will guide the continued evolution of stream processing technologies in the Cloud Native era. This foundational work not only demonstrates the feasibility of a Cloud Native stream processing engine but also sets the stage for extensive future development and enhancements.

# References


Apache. (2023). *Apache kafka.* Retrieved 2024-04-04, from https://kafka.apache.org/37/documentation/streams/architecture

*Apache Flink®.* (2023). Retrieved 2023-11-03, from https://flink.apache.org/

*client-go.* (2023, November). Kubernetes. Retrieved 2023-11-03, from https://github.com/kubernetes/client-go (original-date: 2016-08-25T00:19:38Z)

CNCF. (2015, June). *New Cloud Native Computing Foundation to drive alignment among container technologies.* Retrieved 2024-04-04, from https://www.cncf.io/announcements/2015/06/21/new-cloud-native-computing-foundation-to-drive-alignment-among-container-technologies/

Dobies, J., & Wood, J. (2020). *Kubernetes Operators: Automating the Container Orchestration Platform* (First ed.). O'Reilly.

*Flink Kubernetes Operator Github.* (2023, November). The Apache Software Foundation. Retrieved 2023-11-03, from https://github.com/apache/flink-kubernetes-operator (original-date: 2022-02-10T05:00:35Z)

Fragkoulis, M., Carbone, P., Kalavri, V., & Katsifodimos, A. (2023, January). *A Survey on the Evolution of Stream Processing Systems.* arXiv. Retrieved 2023-11-03, from http://arxiv.org/abs/2008.00842 (arXiv:2008.00842 [cs])

*Function.fission.io/v1.* (2023). Retrieved 2023-11-03, from https://doc.crds.dev/github.com/fission/fission/fission.io/Function/v1

Gannon, D., Barga, R., & Sundaresan, N. (2017). Cloud-native applications. *IEEE Cloud Computing*, *4*(5), 16-21. doi: 10.1109/MCC.2017.4250939

Hausenblas, M., & Schimanski, S. (2019). *Programming Kubernetes: Developing with Cloud-Native Applications* (First ed.). O'Reilly.

Kubernetes. (2024). *Horizontal Pod Autoscaling.* Retrieved 2024-04-04, from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (Section: docs)

*Kubernetes Components.* (2023). Retrieved 2023-11-03, from https://kubernetes.io/docs/concepts/overview/components/ (Section: docs)

*Kubernetes Informer pkg.* (2024). Retrieved 2024-04-04, from https://pkg.go.dev/k8s.io/client-go/informers#SharedInformerFactory

Michels, M. (2022). *FLIP-271: Autoscaling - Apache Flink - Apache Software Foundation.* Retrieved 2024-04-04, from https://cwiki.apache.org/confluence/

display/FLINK/FLIP-271%3A+Autoscaling

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., ... Stoica, I. (2018). *Ray: A distributed framework for emerging ai applications.*

*Operator capabilities.* (2024). Retrieved from https://operatorframework.io/operator-capabilities/

*Operator Framework.* (2023). Retrieved 2023-11-03, from https://operatorframework.io/

*OperatorHub.io.* (2023). Retrieved 2023-11-03, from https://operatorhub.io/

Phillips, B. (2016, November). *Introducing Operators: Putting Operational Knowledge into Software.* Retrieved 2023-11-03, from https://web.archive.org/web/20170129131616/https://coreos.com/blog/introducing-operators.html

Prometheus. (2024). *prometheus-community/helm-charts: Prometheus community Helm charts.* Retrieved 2024-04-04, from https://github.com/prometheus-community/helm-charts

Sayfan, G. (2023). *Mastering Kubernetes* (Fourth ed.). Packt.

Shapira, G., & Sax, M. (2018). *Deploying Kafka Streams Applications with Docker and Kubernetes.* Retrieved 2024-04-04, from https://www.confluent.io/kafka-summit-sf18/deploying-kafka-streams-applications/

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., ... Stoica, I. (2016, oct). Apache spark: a unified engine for big data processing. *Commun. ACM*, *59*(11), 56–65. Retrieved from https://doi.org/10.1145/2934664 doi: 10.1145/2934664

# A Appendix

```
apiVersion: invokeroperator.invoker.io/v1alpha1
kind: InvokerDeployment
metadata:
  labels:
    app.kubernetes.io/name: invokerdeployment
    app.kubernetes.io/instance: invokerdeployment-sample
    app.kubernetes.io/part-of: invoker-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: invoker-operator
  name: sample
spec:
  imagePullPolicy: "IfNotPresent"
  statefulEntities:
    - executorImage: "ruohang/executor-word-count:testv1.1"
      ioImage: "ruohang/iosidecar:testv1.1"
      stateImage: "ruohang/statesidecar:testv1.1"
      ioAddress: "host.docker.internal:9092"
      inputTopic: "test"
      outputTopic: "count"
      name: "word-count"
      pods:
       - name: "word-count-1"
         partitions:
          - "0"
          - "1"
         resources:
          requests:
            memory: "64Mi"
            cpu: "250m"
          limits:
            memory: "256Mi"
            cpu: "500m"
  version: "testv1"
  historyLimit: 3
```

Figure A1: Instance of Invoker Deployment defined by the developer

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sum
  labels:
    app: executor
spec:
  containers:
    - name: executor-pod
      image: "ruohang/executor-sum:testv1.1"
      imagePullPolicy: "Always"
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
        - name: uds-volume
          mountPath: /tmp
    - name: io-sidecar
      image: "ruohang/iosidecar:testv1.1"
      imagePullPolicy: "Always"
      env:
        - name: POD_NAME
          value: "sum"
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
        - name: uds-volume
          mountPath: /tmp
    - name: state-sidecar
      image: "ruohang/statesidecar:testv1.1"
      imagePullPolicy: "Always"
      volumeMounts:
        - name: uds-volume
          mountPath: /tmp
  volumes:
    - name: config-volume
      configMap:
        name: my-configmap
    - name: uds-volume
      emptyDir: { }
```

Figure A2: Executor instance in Pod form as a YAML definition