Lab 12: Deploying a Standalone Flink Cluster on Kubernetes

**Goal:** Deploy a basic, standalone Flink cluster on a local Minikube instance using Kubernetes manifests. You will define the cluster components (JobManager, TaskManager) as Kubernetes objects and access the Flink UI to submit a job.

# INDEX

[**Purpose of this Lab**](#_sz8v0j97ut6e) **3**

[**Prerequisites**](#_vhwhg1hftdyc) **3**

[**Project Structure**](#_qjihjepi6kzp) **3**

[**Part 1: Environment Setup**](#_ng2q7wwjarwa) **4**

[Step 1: Start Minikube](#_4fhb8w74busv) 4

[**Part 2: Defining the Flink Cluster Manifests**](#_1mx1vijbgrqy) **5**

[Step 1: Create the Project Directory](#_t1ohwznxt19) 6

[Step 2: Create the Flink ConfigMap](#_gohghil9ys0o) 6

[Step 3: Create the JobManager Service](#_4msi4feb4tin) 8

[Step 4: Create the JobManager Deployment](#_s1dn3fmkruxl) 8

[Step 5: Create the TaskManager Deployment](#_bmwr1hufytvq) 10

[**Part 3: Deployment and Execution**](#_wxi6di976rml) **12**

[Step 1: Apply the Kubernetes Manifests](#_134hc1lgjygr) 12

[Step 2: Verify the Cluster Components](#_pcwt913iow41) 12

[**Part 4: Verification and Observation**](#_p68jsh5ziop6) **13**

[Step 1: Access the Flink Web UI](#_su53hub6p1pn) 13

[Step 2: Submit a Sample Job](#_za78gpp958ev) 13

[**Part 5: Understanding the Deployment and Next Steps**](#_j7tme07mvcnt) **14**

[**Part 6: Cleanup**](#_55rkr09ptk3l) **15**

[Step 1: Delete Flink Cluster Resources](#_xkzil49a9lvm) 15

[Step 2: Stop Minikube](#_ljwzfcakpwz3) 15

# 

# Purpose of this Lab

This lab provides a foundational understanding of running Apache Flink natively on Kubernetes without using a specialized operator. You will manually define and deploy each component of a Flink cluster, giving you a clear picture of the underlying resources involved.

By the end of this lab, you will:

* **Decouple Configuration:** Use a Kubernetes ConfigMap to manage Flink's configuration externally from the container images.
* **Define Cluster Components:** Create Deployment manifests for Flink's JobManager and TaskManager processes.
* **Enable Network Access:** Expose the JobManager's UI and RPC ports using a Kubernetes Service.
* **Deploy and Verify:** Apply the manifests to a Minikube cluster and confirm that all components are running correctly.
* **Interact with the Cluster:** Use port-forwarding to access the Flink Web UI and submit an example job from your local machine.

# Prerequisites

To successfully complete this lab, the following are required:

* **Local Kubernetes Environment:** A functioning local cluster setup using Minikube, with kubectl and a container runtime (e.g., Docker) properly installed and configured.
* **Apache Flink Distribution:** A local installation of Apache Flink is necessary for using the command-line interface (CLI) to submit jobs. The $FLINK\_HOME environment variable must be set to the root of the Flink installation directory.
* **Kubernetes CLI Familiarity:** A working knowledge of fundamental kubectl commands to interact with cluster resources (e.g., apply, get, delete, port-forward).
* **Text Editor/IDE:** A code editor for creating and modifying YAML manifest files, such as VS Code or similar.

# Project Structure

At the end of this lab, you will have a self-contained project directory with all the necessary Kubernetes resource definitions:

|  |
| --- |
| ~/flink-k8s-lab/ ├── flink-configuration-configmap.yaml ├── jobmanager-service.yaml ├── jobmanager-deployment.yaml └── taskmanager-deployment.yaml |

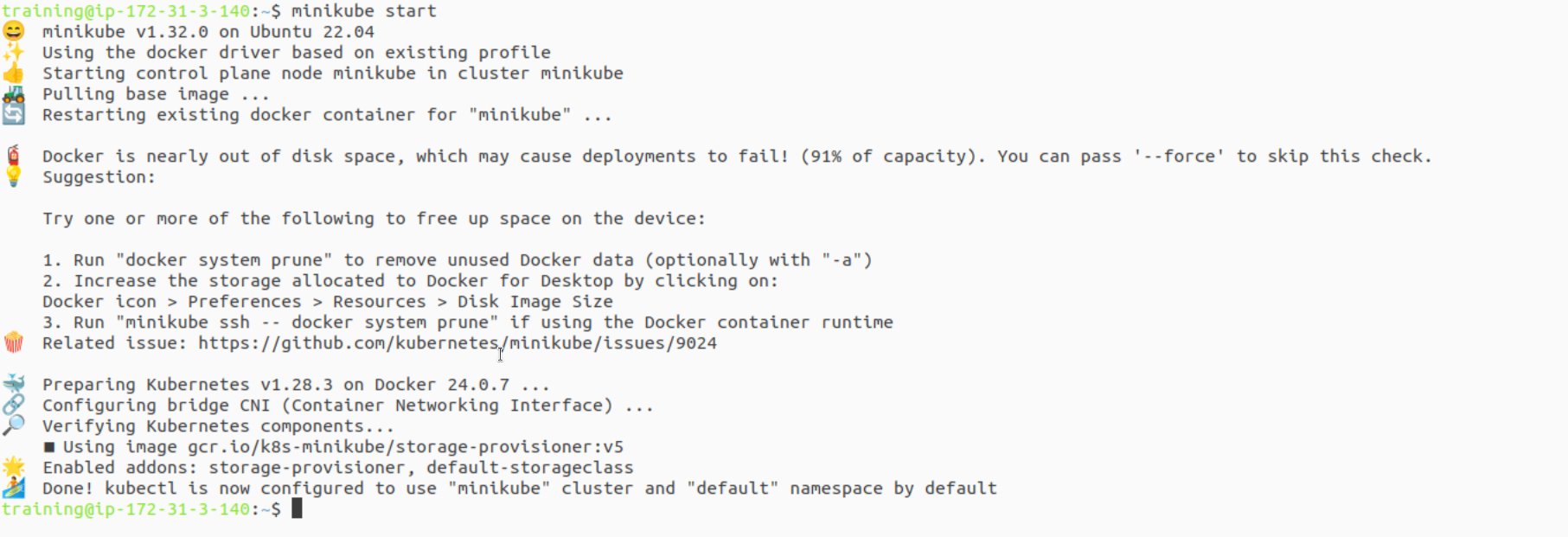
# Part 1: Environment Setup

First, we need to start our local Kubernetes cluster using Minikube.

## Step 1: Start Minikube

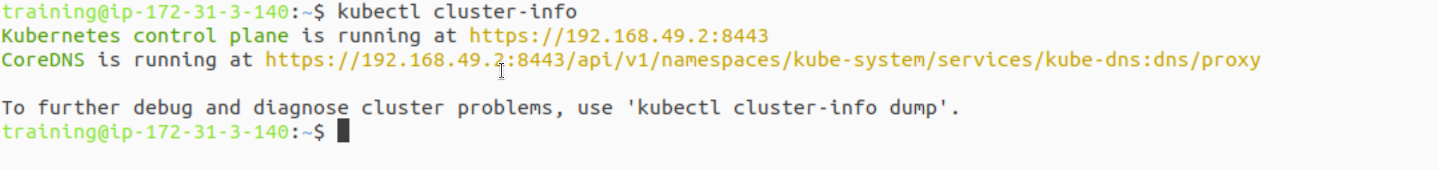
Open your terminal and run the start command:

|  |
| --- |
| minikube start |

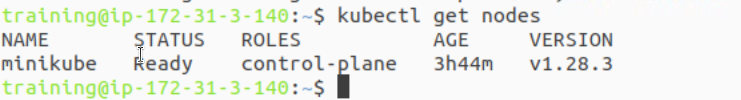


Once it completes, verify that your kubectl context is correctly pointing to the Minikube cluster.

|  |
| --- |
| kubectl cluster-info |



|  |
| --- |
| kubectl get nodes |



You should see output confirming that the Minikube control plane is running and your single node has a Ready status.

**Step 2: Configure Minikube Networking** To ensure Flink's components can communicate with each other through Kubernetes services within the Minikube environment, you must enable promiscuous mode on the Docker network interface.

Execute the following command:

|  |
| --- |
| minikube ssh 'sudo ip link set docker0 promisc on' |



This allows the network interface to pass traffic that it would otherwise discard, enabling the necessary service-based discovery for Flink.

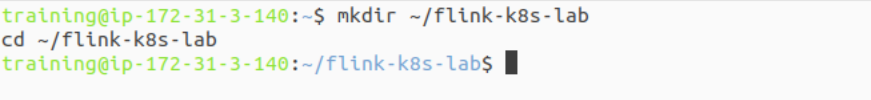
# Part 2: Defining the Flink Cluster Manifests

In this part, we will create the four YAML files that define our Flink cluster.

## Step 1: Create the Project Directory

Let's create a dedicated directory for our lab files.

|  |
| --- |
| mkdir ~/flink-k8s-lab cd ~/flink-k8s-lab |



## Step 2: Create the Flink ConfigMap

A ConfigMap is a Kubernetes object used to store non-confidential data in key-value pairs. We will use it to supply the flink-conf.yaml and logging properties, allowing us to manage configuration without rebuilding our container image.

Create the file:

|  |
| --- |
| code flink-configuration-configmap.yaml |

Paste the following YAML. Note how jobmanager.rpc.address is set to flink-jobmanager, which will be the name of our JobManager Service.

|  |
| --- |
| apiVersion: v1 kind: ConfigMap metadata:  name: flink-config  labels:  app: flink data:  config.yaml: |+  jobmanager.rpc.address: flink-jobmanager  taskmanager.numberOfTaskSlots: 2  blob.server.port: 6124  jobmanager.rpc.port: 6123  taskmanager.rpc.port: 6122  jobmanager.memory.process.size: 1600m  taskmanager.memory.process.size: 1728m  parallelism.default: 2   log4j-console.properties: |+  # This affects logging for both user code and Flink  rootLogger.level = INFO  rootLogger.appenderRef.console.ref = ConsoleAppender  rootLogger.appenderRef.rolling.ref = RollingFileAppender   # The following lines keep the log level of common libraries/connectors on  # log level INFO. The root logger does not override this. You have to manually  # change the log levels here.  logger.pekko.name = org.apache.pekko  logger.pekko.level = INFO  logger.kafka.name= org.apache.kafka  logger.kafka.level = INFO  logger.hadoop.name = org.apache.hadoop  logger.hadoop.level = INFO  logger.zookeeper.name = org.apache.zookeeper  logger.zookeeper.level = INFO   # Log all infos to the console  appender.console.name = ConsoleAppender  appender.console.type = CONSOLE  appender.console.layout.type = PatternLayout  appender.console.layout.pattern = %d{yyyy-MM-dd HH:mm:ss,SSS} %-5p %-60c %x - %m%n   # Suppress the irrelevant (wrong) warnings from the Netty channel handler  logger.netty.name = org.jboss.netty.channel.DefaultChannelPipeline  logger.netty.level = OFF |

## Step 3: Create the JobManager Service

A Kubernetes Service provides a stable network endpoint (a DNS name and IP address) to access a set of Pods. We need one so the TaskManagers and the outside world can reliably communicate with the JobManager, even if its Pod restarts.

Create the file:

|  |
| --- |
| code jobmanager-service.yaml |

Paste the following YAML. This ClusterIP service creates a stable internal DNS name (flink-jobmanager) that other pods in the cluster can use.

|  |
| --- |
| apiVersion: v1 kind: Service metadata:  name: flink-jobmanager spec:  type: ClusterIP  ports:  - name: rpc  port: 6123  - name: blob-server  port: 6124  - name: webui  port: 8081  selector:  app: flink  component: jobmanager |

## Step 4: Create the JobManager Deployment

A Deployment manages a set of replica Pods, ensuring that a specified number of them are always running. This manifest defines the JobManager Pod.

Create the file:

|  |
| --- |
| code jobmanager-deployment.yaml |

Paste the following YAML. Key things to notice:

* replicas: 1: We only need one JobManager.
* image: We use an official Flink image.
* args: ["jobmanager"]: This command starts the JobManager process inside the container.
* volumeMounts and volumes: This section mounts our ConfigMap as files into the /opt/flink/conf directory, overriding the default configurations.

|  |
| --- |
| apiVersion: apps/v1 kind: Deployment metadata:  name: flink-jobmanager spec:  replicas: 1  selector:  matchLabels:  app: flink  component: jobmanager  template:  metadata:  labels:  app: flink  component: jobmanager  spec:  containers:  - name: jobmanager  image: apache/flink:1.19.0-scala\_2.12  args: ["jobmanager"]  ports:  - containerPort: 6123  name: rpc  - containerPort: 6124  name: blob-server  - containerPort: 8081  name: webui  livenessProbe:  tcpSocket:  port: 6123  initialDelaySeconds: 30  periodSeconds: 60  volumeMounts:  - name: flink-config-volume  mountPath: /opt/flink/conf  securityContext:  runAsUser: 9999 # refers to user 'flink' from official flink image  volumes:  - name: flink-config-volume  configMap:  name: flink-config  items:  - key: config.yaml  path: flink-conf.yaml  - key: log4j-console.properties  path: log4j-console.properties |

## Step 5: Create the TaskManager Deployment

Finally, we define the Deployment for our worker processes, the TaskManagers. The structure is very similar to the JobManager's.

Create the file:

|  |
| --- |
| code taskmanager-deployment.yaml |

Paste the following YAML. The main differences are:

* replicas: 2: We are starting with two TaskManagers.
* args: ["taskmanager"]: This starts the TaskManager process.
* It also mounts the same ConfigMap to ensure its configuration is consistent with the JobManager.

|  |
| --- |
| apiVersion: apps/v1 kind: Deployment metadata:  name: flink-taskmanager spec:  replicas: 2  selector:  matchLabels:  app: flink  component: taskmanager  template:  metadata:  labels:  app: flink  component: taskmanager  spec:  containers:  - name: taskmanager  image: apache/flink:1.19.0-scala\_2.12  args: ["taskmanager"]  ports:  - containerPort: 6122  name: rpc  livenessProbe:  tcpSocket:  port: 6122  initialDelaySeconds: 30  periodSeconds: 60  volumeMounts:  - name: flink-config-volume  mountPath: /opt/flink/conf/  securityContext:  runAsUser: 9999 # refers to user 'flink' from official flink image  volumes:  - name: flink-config-volume  configMap:  name: flink-config  items:  - key: config.yaml  path: flink-conf.yaml  - key: log4j-console.properties  path: log4j-console.properties |

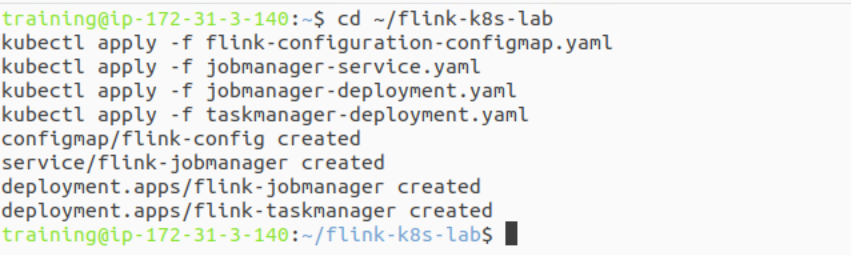
# Part 3: Deployment and Execution

With our manifests defined, we can now apply them to the Kubernetes cluster.

## Step 1: Apply the Kubernetes Manifests

Run the following kubectl apply commands from your ~/flink-k8s-lab directory. It's good practice to apply the ConfigMap and Service first, as the Deployments depend on them.

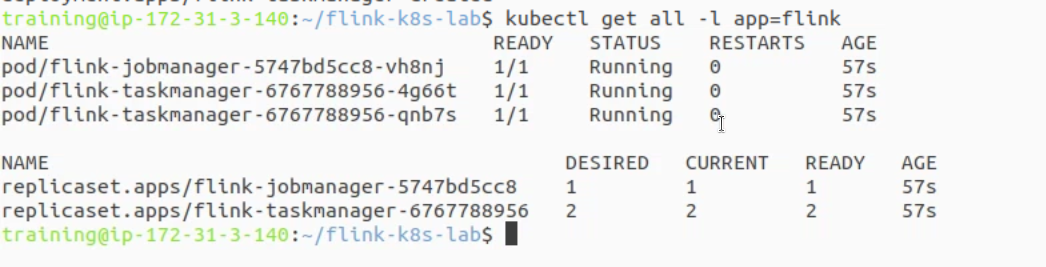
|  |
| --- |
| cd ~/flink-k8s-lab kubectl apply -f flink-configuration-configmap.yaml kubectl apply -f jobmanager-service.yaml kubectl apply -f jobmanager-deployment.yaml kubectl apply -f taskmanager-deployment.yaml |



## Step 2: Verify the Cluster Components

Check the status of all the resources you just created.

|  |
| --- |
| kubectl get all -l app=flink |



Wait a minute for the pods to be downloaded and started. You should see one flink-jobmanager pod and two flink-taskmanager pods, all with a status of Running. You will also see the corresponding Deployments, ReplicaSets, and the Service.

# Part 4: Verification and Observation

Let's confirm the cluster is fully operational by accessing the UI and submitting a job.

## Step 1: Access the Flink Web UI

To access the UI from your local machine, we will use kubectl to forward a local port to the JobManager pod's web UI port (8081).

In an new Terminal window run the following command :

|  |
| --- |
| kubectl port-forward deployment/flink-jobmanager 8081:8081 |

Now, open your web browser and navigate to **http://localhost:8081**.

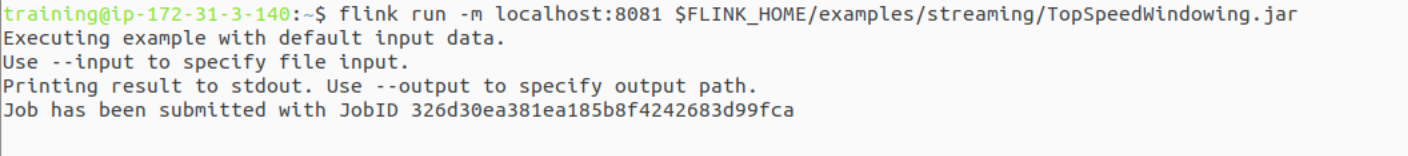
You should see the Flink Dashboard. On the overview page, check that you have **4 Total Task Slots** (2 TaskManagers x 2 slots each, as defined in our ConfigMap).

Keep this terminal window in order to access the UI.

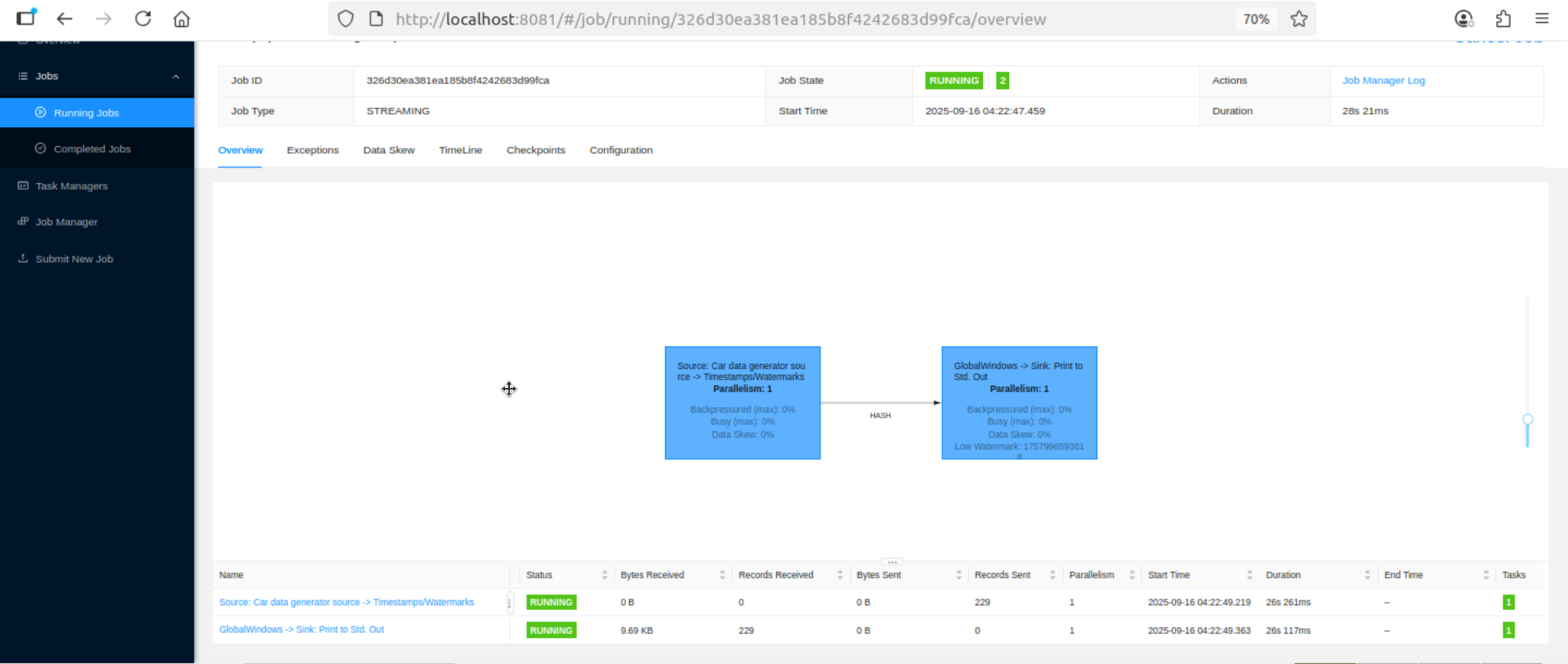
## Step 2: Submit a Sample Job

Keep the port-forward running. Open a new terminal and use the Flink CLI to submit one of the example jobs. The -m localhost:8081 flag tells the CLI where to find the JobManager.

|  |
| --- |
| flink run -m localhost:8081 $FLINK\_HOME/examples/streaming/TopSpeedWindowing.jar |



After the job submits, check the Flink UI again. You will see the "Top Speed Windowing" job in the "Running Jobs" list. Click on it to explore its execution graph and metrics.



# Part 5: Understanding the Deployment and Next Steps

You have successfully deployed a standalone Flink cluster on Kubernetes. This setup provides a solid understanding of the core components but is not ideal for dynamic production workloads. Here's why and what to do next.

* **Scaling TaskManagers:** Our setup is static. You can easily scale the number of workers up or down manually with a single command, which is a great feature of Kubernetes.  
  # Scale up to 4 TaskManagers

|  |
| --- |
| kubectl scale deployment/flink-taskmanager --replicas=4 |

* **State and Persistence:** This cluster is stateless. If a TaskManager pod dies, any state held in its memory (like window calculations) is lost. For fault-tolerant applications, you must configure a checkpointing directory using a persistent storage solution, such as a Kubernetes PersistentVolume.
* **Production Deployments:** For production use cases, especially those requiring dynamic resource allocation per job, the manual approach is cumbersome. The **Flink Kubernetes Operator** is the recommended solution. It extends the Kubernetes API to manage the entire Flink application lifecycle natively, automating deployment, configuration, scaling, and recovery.

# Part 6: Cleanup

To clean up the resources created in this lab, delete them using the manifest files and then stop Minikube.

## Step 1: Delete Flink Cluster Resources

Run these commands from your ~/flink-k8s-lab directory:

|  |
| --- |
| cd ~/flink-k8s-lab kubectl delete -f taskmanager-deployment.yaml kubectl delete -f jobmanager-deployment.yaml kubectl delete -f jobmanager-service.yaml kubectl delete -f flink-configuration-configmap.yaml |

## Step 2: Stop Minikube

|  |
| --- |
| **minikube stop** |