KubeCon CloudNativeCon

THE LINUX FOUNDATION

OPEN SOURCE SUMMIT

AI_dev

Open Source GenAI & ML Summit

China 2024

Scaling Kubernetes: Best Practices for Managing Large-Scale Batch Jobs with Spark and Argo Workflow

Yu Zhuang Alibaba Cloud
Jiaxu Liu Alibaba Cloud

About Us

Alibaba Cloud Container Kubernetes Service (ACK)

Alibaba Cloud

Yu Zhuang:

- 9 years on Kubernetes
- ACK Architect
- Focus on offline batch job and multi-cluster

Jiaxu Liu:

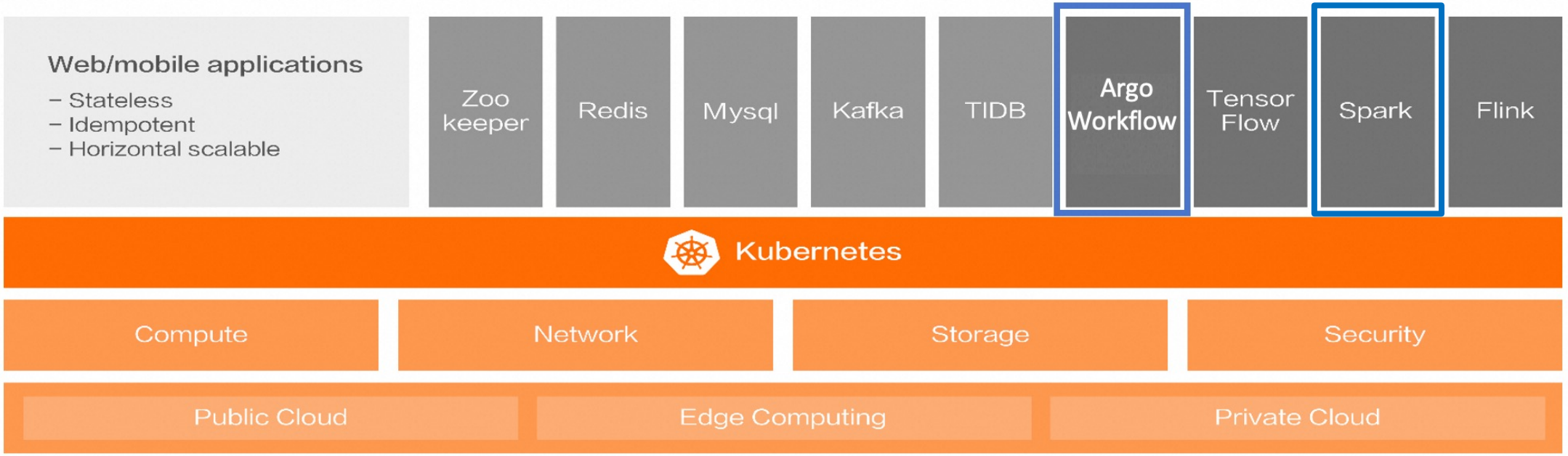
- 7 years on Kubernetes
- ACK Senior Engineer
- Focus on observability and large-scale cluster management

Agenda

1. Why running offline batch job on Kubernetes
2. Spark on Kubernetes
3. Argo Workflow on Kubernetes
4. Best practices of large scale offline batch job on Kubernetes

Cloud Native OS - Kubernetes

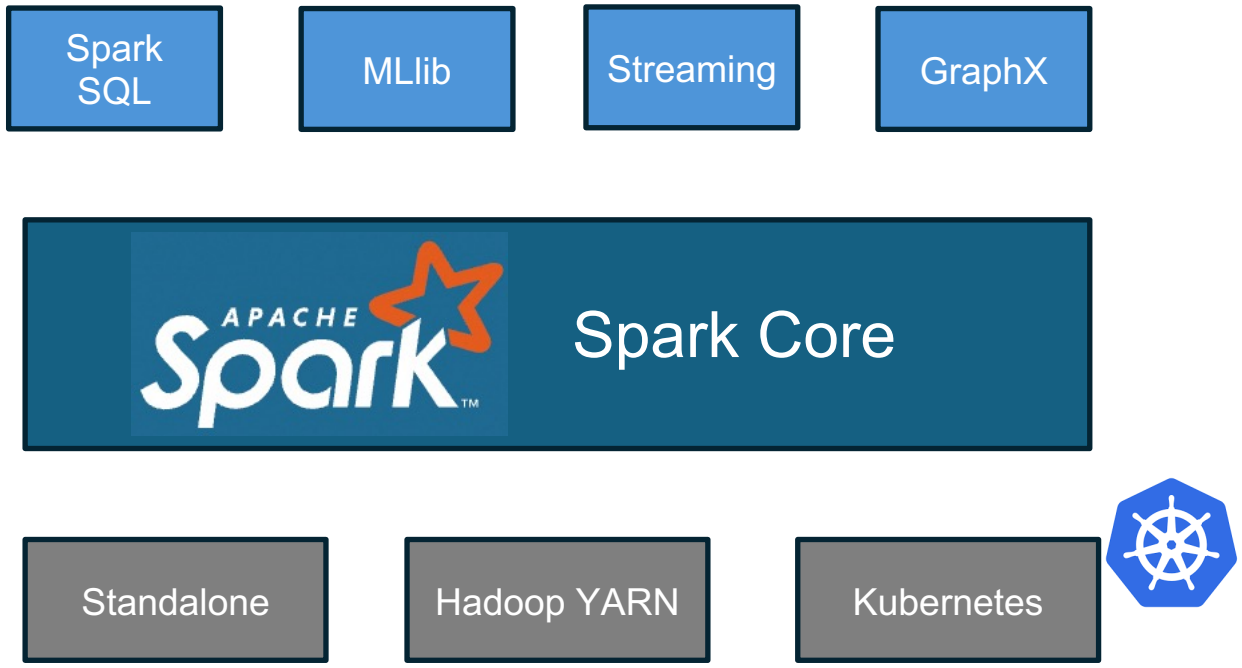
Stateless Application, Stateful Application, Batch Jobs (Data Process, AI training/Inference, Science Computing)



Why run batch jobs on Kubernetes

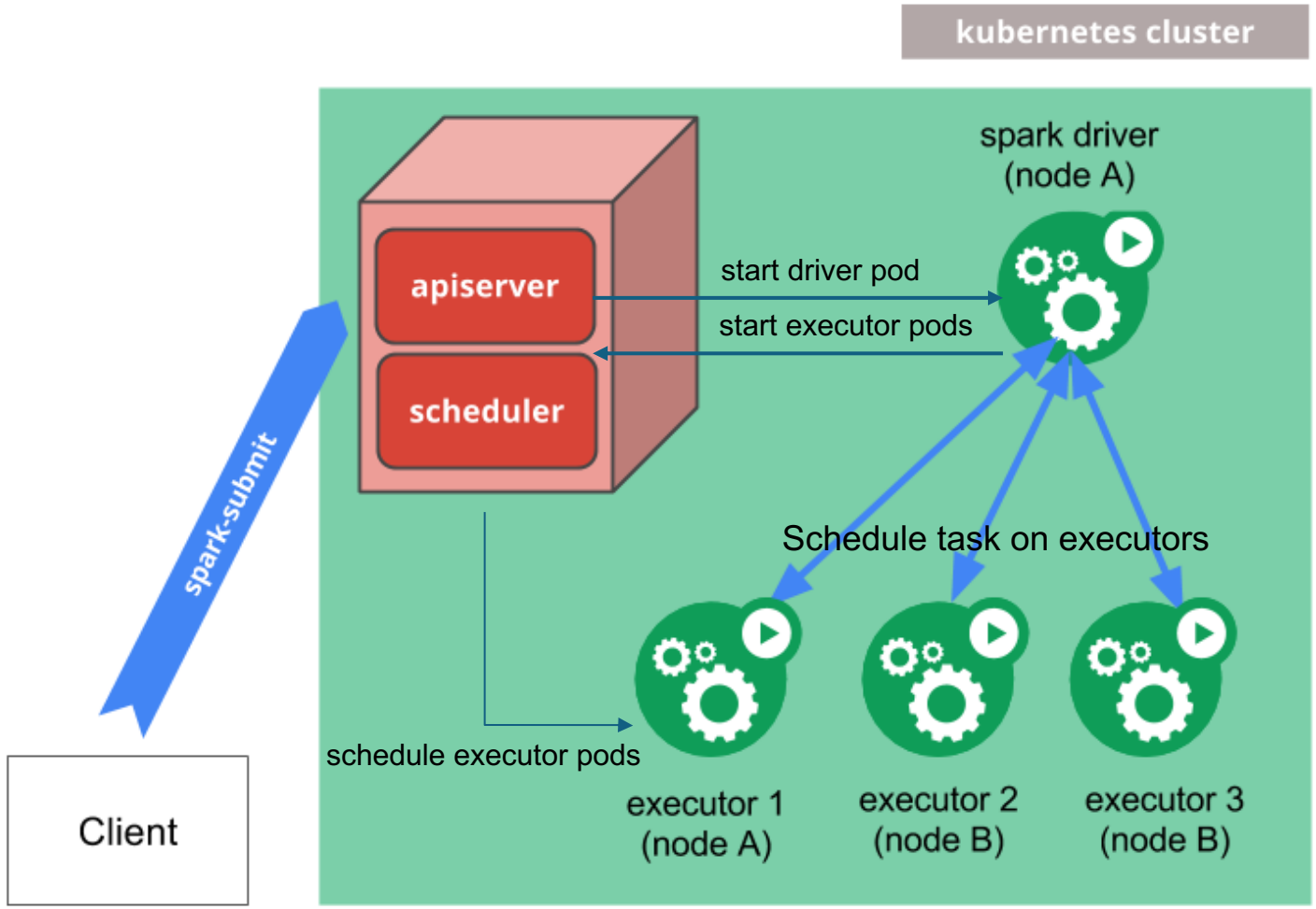
- Single infrastructure: manages both online and offline workloads.
- Scalability: scales your offline jobs horizontally.
- Cost Optimization: shares resource in on premises and run on serverless in cloud.
- Multi-Tenancy: isolates with namespace, resource quota, and kqueue.
- Portability: allows to migrate offline jobs in different cloud providers.
- Ecosystem: provides operation capabilities, e.g. monitoring, logging, and security.

Spark



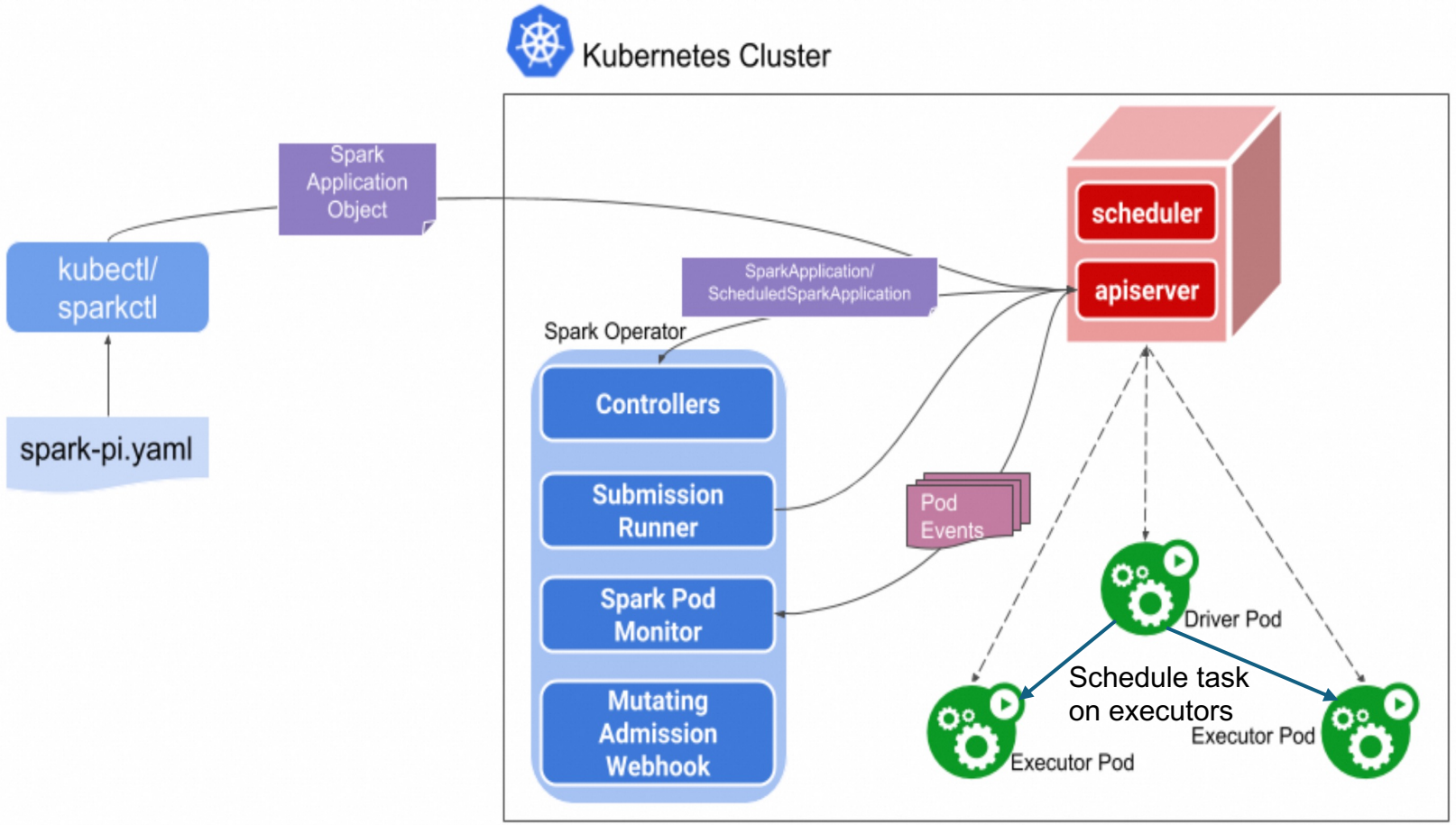
Apache Spark is a unified analytics engine for large-scale data processing. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, pandas API on Spark for pandas workloads, MLlib for machine learning, GraphX for graph processing, and Structured Streaming for incremental computation and stream processing.

Spark on Kubernetes




```
./bin/spark-submit \
--master k8s://https://<k8s-apiserver-host>:<k8s-apiserver-port> \
--deploy-mode cluster \
--name spark-pi \
--class org.apache.spark.examples.SparkPi \
--conf spark.executor.instances=3 \
--conf spark.kubernetes.container.image=<spark-image> \
local:///path/to/examples.jar
```

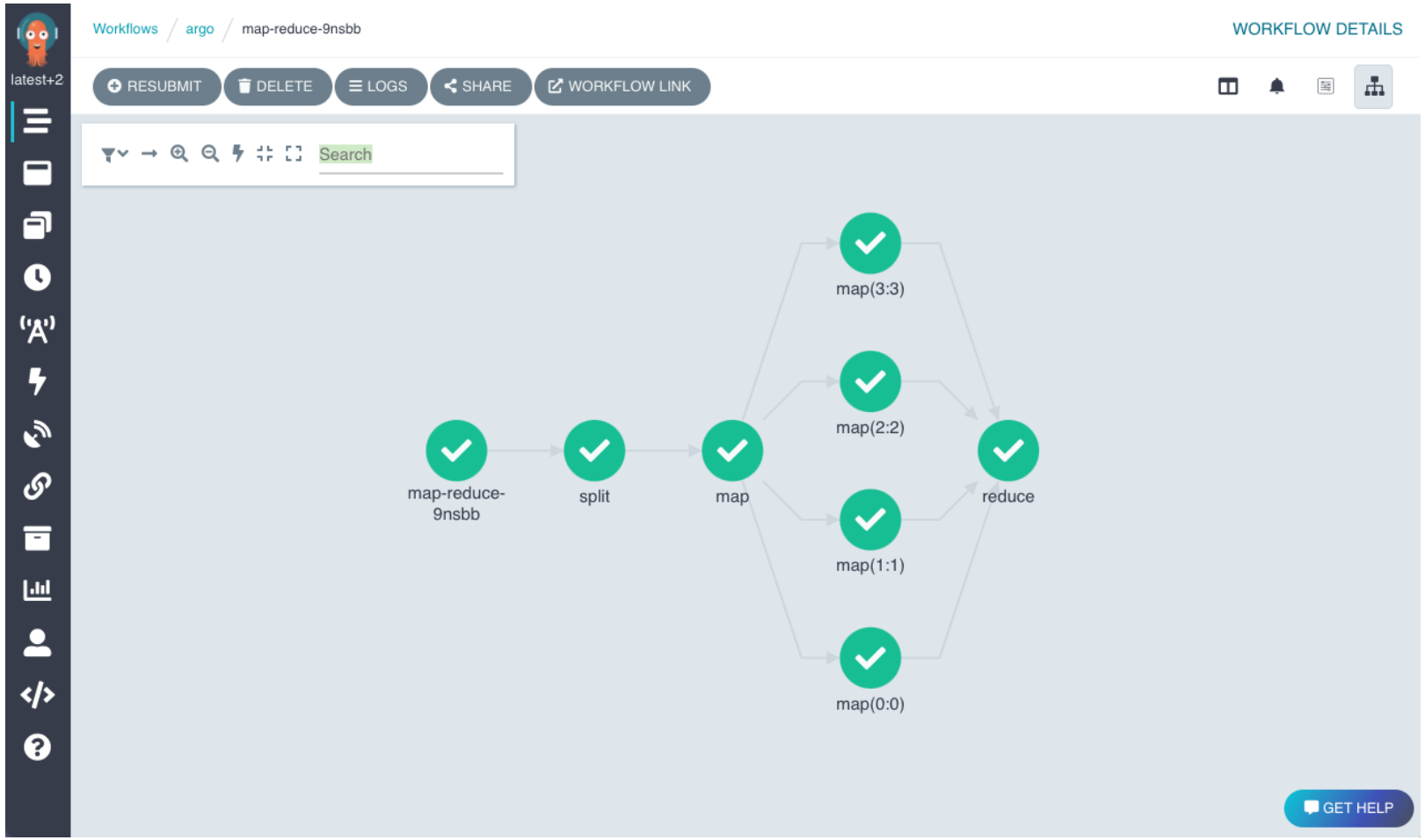
Spark on Kubernetes




```
apiVersion: sparkoperator.k8s.io/v1beta2
kind: SparkApplication
metadata:
  name: spark-pi
  namespace: default
spec:
  type: Scala
  mode: cluster
  image: spark:3.5.0
  imagePullPolicy: IfNotPresent
  mainClass: org.apache.spark.examples.SparkPi
  mainApplicationFile: local:///opt/spark/examples/jars/spark-
examples_2.12-3.5.0.jar
  sparkVersion: 3.5.0
  driver:
    labels:
      version: 3.5.0
    cores: 1
    coreLimit: 1200m
    memory: 512m
    serviceAccount: spark-operator-spark
  executor:
    labels:
      version: 3.5.0
    instances: 1
    cores: 1
    coreLimit: 1200m
    memory: 512m
```

Argo Workflows -- Graduated

General Container-native workflow engine for orchestrating parallel jobs on Kubernetes



Use Cases

- Data processing
- Scientific computing
- Machine learning
- CI/CD

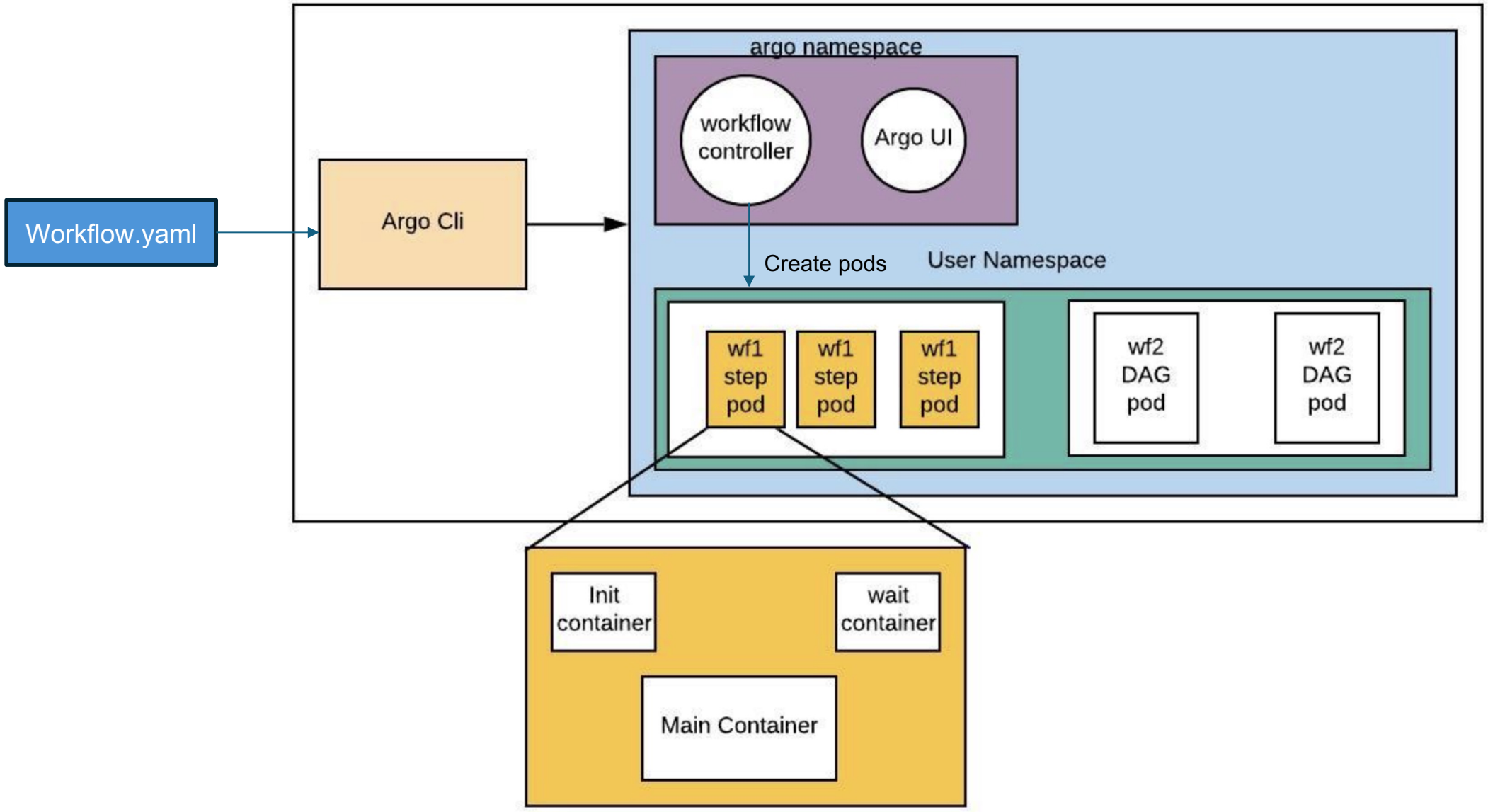
Workflow Definition


```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: hello-world-
spec:
  entrypoint: hello-world-example
  templates:
  - name: hello-world-example
    steps:
    - - name: generate-artifact
        template: whalesay
  - name: whalesay
    container:
      image: docker/whalesay:latest
      command: [sh, -c]
      args: ["cowsay hello world | tee /tmp/hello_world.txt"]
    outputs:
      artifacts:
      - name: hello-art
        path: /tmp/hello_world.txt
status:
  nodes: hello-world-example-6ftnv
  taskResultsCompletionStatus: hello-world-example-6ftnv: false
  phase: Running
```

- Template definition: sequence or complex DAG
- Template: image, argument, inputs and outputs
- Status: nodes, phase, taskResultsCompletionStatus

Also Support Python SDK

Argo Workflows on Kubernetes



Challenge for running Spark and Argo Workflow on Kubernetes

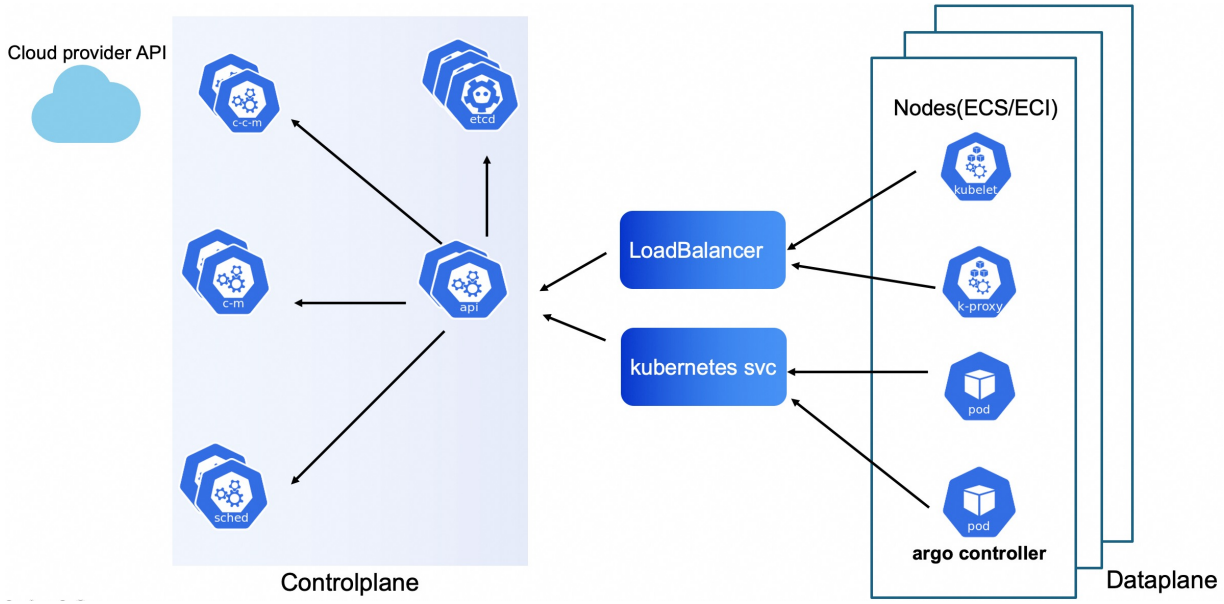
1. Large scale pods creating and deleting frequently for parallel computing
   a) Thousands of pods in one workflow step.
   b) Thousands of executor pod in one spark application.
2. Huge update when pod creating/deleting, like svc/endpoints, env injection, other controller.
3. Manifest could be too large for etcd.
4. High cost of compute resource.

Best practices for controller

1. Config GC properly to avoid resource accumulation over ETCD limitation.
2. Offload large workflow to RDS.
3. Increase qps and burst.
4. Use spot and serverless container to save cost.

Kubernetes Overview

Controlplane, dataplane, and cloud resources are an integrally interconnected whole!
If any component or path becomes a bottleneck, it may negatively impact the overall stability and performance of the cluster.



Controlplane

- Central management of a Kubernetes cluster, responsible for maintaining the desired state of the cluster and orchestrating the various worker nodes.

Includes api-server, etcd, kube-controller-manager, kube-scheduler， cloud-controller-manager, etc.

Dataplane

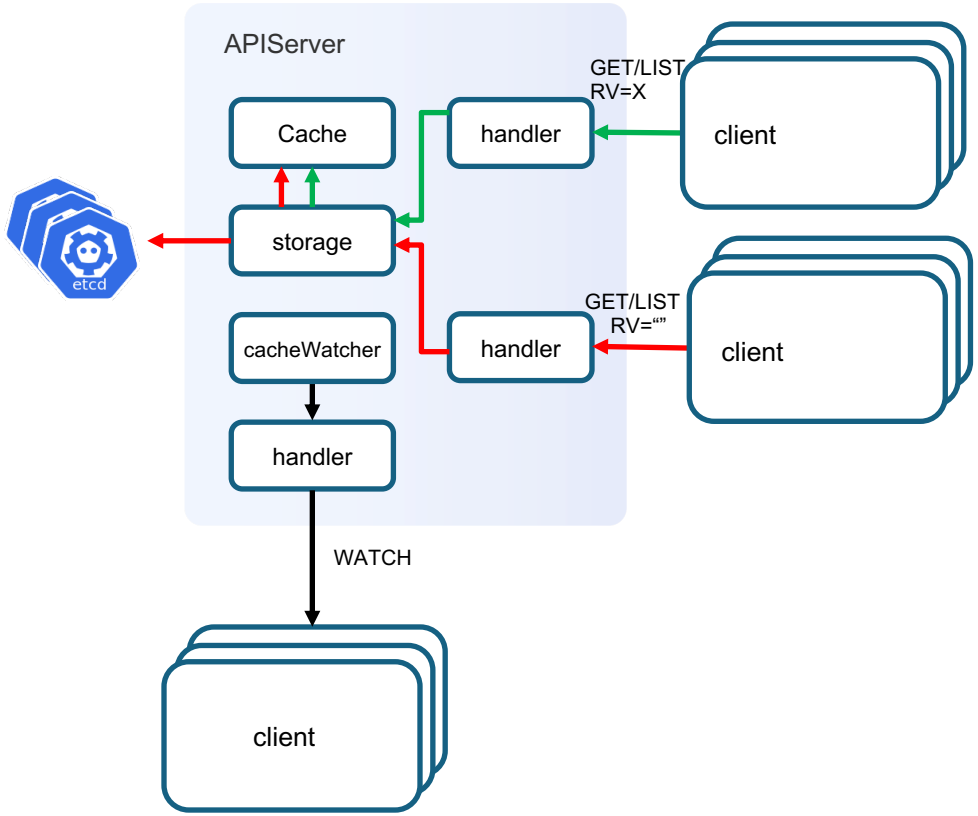
- Handles the processing, transfer, and storage of data among applications and services.

Includes K8s components kubelet/kube-proxy, controllers, like argo workflow controllers, spark operators, etc.

Cloud Resources

- Provides nodes, SLB, and other cloud/physical resources for the K8s cluster

Kubernetes API Requests



K8s resource object has a ResourceVersion (RV), which represents the specific revision of object stored in etcd.

GET/LIST Requests with ResourceVersion (RV=X):

1. If RV in the cache is >= X, return the object in the cache directly;
else, wait for the result within a 3-second timeout period, if RV in the cache is >= X, return the object in the cache;
if timeout, return an error "Too large resource version.”

2. If RV=0, return the object in the cache directly.

GET/LIST Requests without ResourceVersion:

For K8s >=R1.31, ConsistentListFromCache feature gate is enabled, APIServer return object in cache;
For K8s <=R1.30, the APIServer performs a consistent read from etcd (quorum read).

Filtering in etcd:

etcd key structure： /registry/{resource type}/{namespace}/{resource name}
Filtering in etcd is limited to resource/namespace/specific resource.
Any other filtering, such as using labelSelector/fieldSelector, is performed in memory by APIServer.

Informer Mechanism:

Includes LIST (RV=0) + WATCH (RV=X, X comes from the LIST request). The client maintains a local cache that is updated via WATCH notifications from the APIServer, reducing the need for frequent LIST requests. It is recommended to use the Informer approach for accessing APIServer.

Recap

1. For K8s <= 1.30, a LIST request without a ResourceVersion directly queries x the APIServer and etcd, which can place significant pressure on the overall system.
2. Using labelSelector or fieldSelector only filters in the memory of the APIServer and does not alleviate pressure on etcd.
3. The informer optimally combines LIST and WATCH requests to minimize pressure on both APIServer and etcd, making it the recommended approach.

Examples of LIST Requests

Example of LIST Requests

1. LIST api/v1/pods?filedSelector=spec.nodeName%3DworkerNode1

This request performs quorum reads from etcd for K8s <=R1.30.

etcd is merely a KV store and does not have filtering capabilities by labelSelector/fieldSelector (it only handles limit/continue). The APIServer (for K8s <=R1.30) reads the complete data from etcd and then filters it in memory.

2. LIST api/v1/pods?filedSelector=spec.nodeName%3DworkerNode1&resourceVersion=0

The difference from #1 is that it adds **resourceVersion=0**, which allows the apiserver to read data from the cache, resulting in a significant performance improvement, especially in large-scale scenarios.

3. LIST api/v1/pods?limit=500&resourceVersion=0

Although the limit pagination method is used for access, setting resourceVersion=0 causes the apiserver to ignore limit=500. As a result, the client retrieves the data of all pods from the cache.

Best practices for APIServer

APIServer Auto-Scale

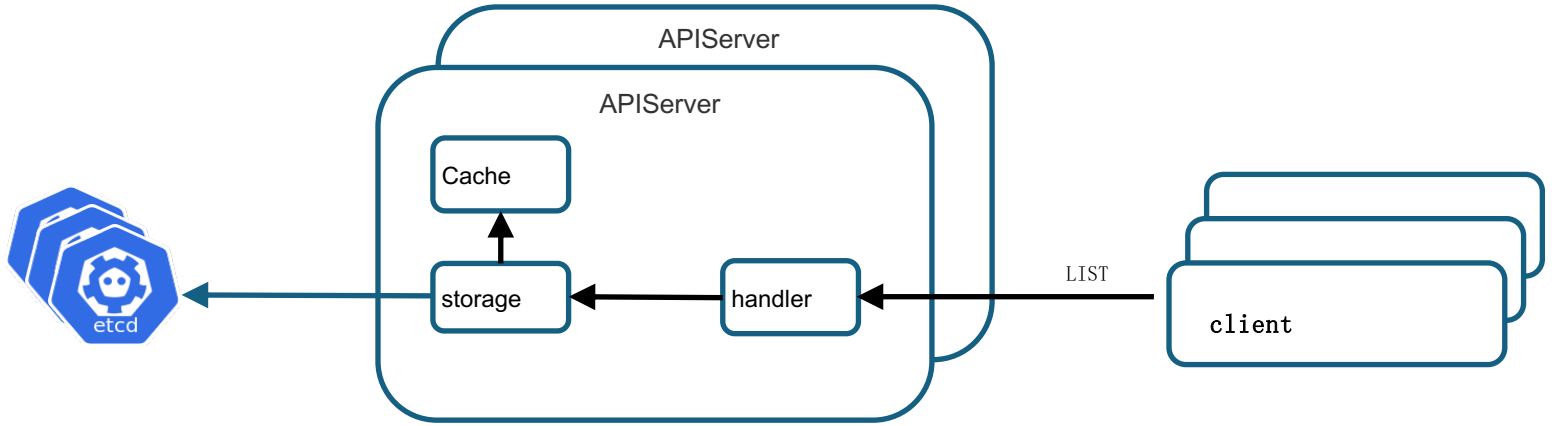
Suggestion: Implements automatic elasticity of APIServer instances based on access pressure and cluster capacity. Includes HPA and VPA.

Effect: Dynamic adaptive elasticity per request pressure.

GoAway

Suggestion: Uneven load can lead to increased resource overhead on individual APIServer instance, making them more susceptible to OOM conditions. The GoAway feature probabilistically disconnects existing TCP connections and establishes new ones, achieving a load balancing effect.

Effect: Stabilize the operating cluster and addresses the issue of load imbalance.



Admission Webhook

Suggestion

Use admission webhook with caution, and monitor webhook's execution time and failure handling.

Because:

Each request to the APIServer incurs additional latency due to the webhook call. If not optimized, this can lead to slow API responses.

If an admission webhook fails (e.g., due to network issues or code bugs), the request may fail and result in disruptions. Proper handling and redundancy strategies are critical.

Effect

Control admission webhook's side effect for the cluster.

Best practices for APIServer

Golang parameter optimization

Suggestion

1. GOGC Environment Variable

Due to adjustment of the Go 1.18 GC threshold algorithm: the memory usage in K8s 1.24 will increase by 25% (data sourced from the community: https://github.com/kubernetes/kubernetes/issues/108357). By using the GOGC parameter, memory resources can be effectively reduced, although CPU consumption may increase.

2. GOMAXPROCS Environment Variable

Configure goroutine concurrency based on the number of CPU cores.

Effect

By tuning GO GC, control the accumulation of memory resources in the APIServer, thereby reducing the possibility of OOM; by controlling concurrency, improve the efficiency of goroutines in the APIServer.

Upgrade K8s to benefit new features

Suggestion

Upgrading K8s introduces security enhancement, bugfix, new features, particularly those for performance and scalability improvement.

Feature gate is used to manage feature enablement, and beta version is enabled by default.

Here are important feature gates for scalability:

Feature Gate	Beta/GA	Feature
ConsistentListFromCache	Beta:R1.31	Ensures consistent API responses by using cached data.
APIPriorityAndFairness	Beta:R1.20 GA:R1.29	High-priority requests are processed preferentially while overall fairness in resource allocation.
EfficientWatchResumption	Beta:R1.21	Optimize the resumption of Kubernetes watch operations.
WatchList	Alpha:R1.27	Requesting the initial state as part of the watch request.
APIListChunking	Beta:R1.9 GA:R1.29	Enable the API clients to retrieve (LIST or GET) resources from API server in chunks.

Effect

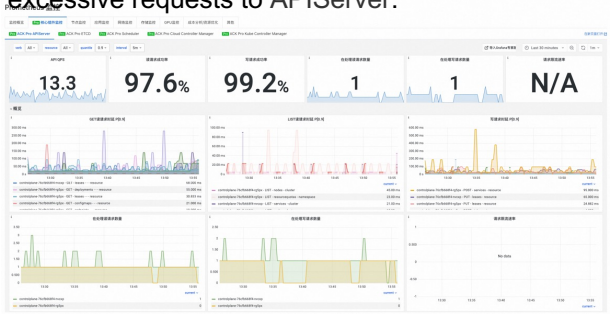
Benefit bugfix, security enhancement, new features particularly for scalability and performance.

Capability to Monitor Cluster

Suggestion

Fully monitor cluster controlplane and dataplane, such as response time, qps, flow control, network traffic, resources, etc. Set up alert system based on monitoring system.

Case : find out components which make excessive requests to APIServer.



Effect

Capability to deeply monitor cluster, including controlplane and dataplane.

Best practice for Client/Server Request Control

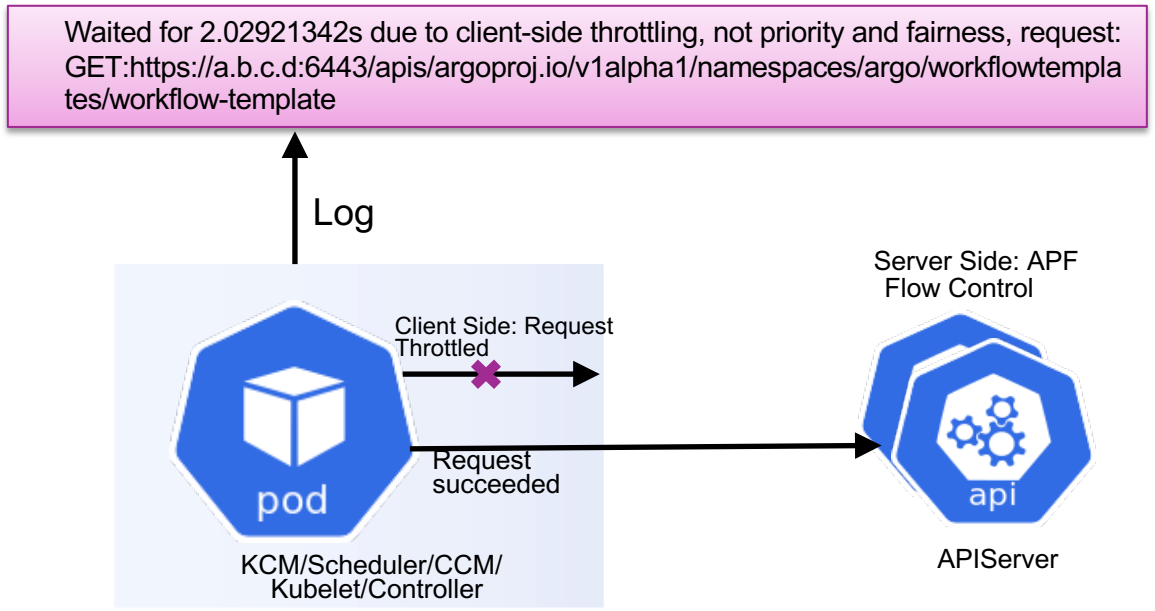
Client-Side qps/burst & APIServer APF Flow Control

Suggestion

Client-Side: tune qps/burst parameters for KCM/Scheduler/CCM/Kubelet/Controller. Especially increase them in large-scale clusters to prevent client-side throttling.
Server-Side: tune APF flow control policies to make sure requests are served as expected by APIServer, monitor the APIServer APF metrics.

Effect

Ensure that the controller can efficiently synchronize data and status through the APIServer to avoid request throttling, and p roper requests are appropriately managed under the control o f APIServer APF.



Best practices for etcd

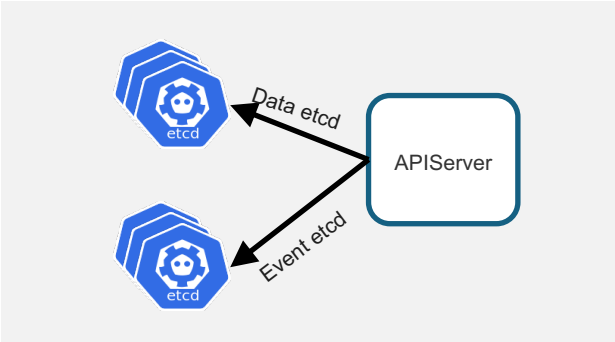
Split etcd for data and events

Suggestion

Data and event are stored in separate etcd clusters.

Effect

Separation of data and event flows eliminates the impact of event flows on data traffic. This reduces the total amount of data stored in a single etcd cluster, thereby improving scalability.



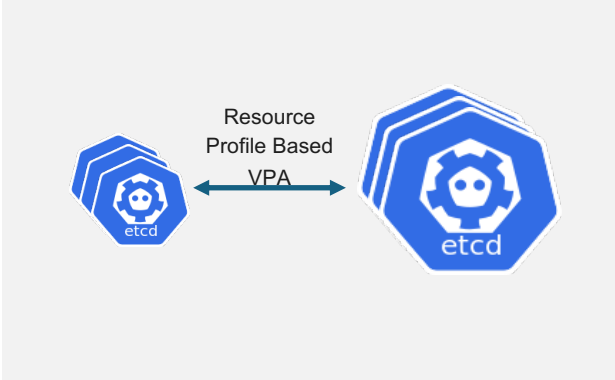
etcd VPA based on resource profile

Suggestion

Dynamically adjust etcd Pod requests/limits based on etcd resource usage and history. Such as component koordinator can help with this automation.

Effect

Dynamic resource capacity adjustment based on actual resource utilization, reduce OOM.



etcd Auto-Defrag

Suggestion

Monitors the etcd cluster's database usage and automatically triggers defragmentation to clean up the database.

Effect

Reduce database size and improve query speed.

Best practices for Cluster Resources

Cluster Resource Control

Suggestion

1. Control the quantity of K8s resources such as Node, Service, Secrets, and ConfigMap, as well as the size of individual resources.

Case: The Prometheus Operator watches all secrets in the cluster. When the number of secrets in a single cluster reaches hundreds of thousands, it results in slow startup for the Operator and consumes a significant amount of memory, reaching several GiB, and may even lead to OOM issues depending on resource limit.

2. Allocate K8s resources across different namespaces (to avoid placing all resources in a single namespace).

Case: A cluster created tens of thousands of Jobs under a single namespace at scheduled intervals without any labels, making it impossible to list and delete them. Ultimately, directly read the Job list from the etcd, then clean up the Jobs based on the Job list.

Effect

Resource isolation, reduce control surface pressure, lower flow transmission.

Best practices for Components

Tune LIST Requests

Suggestion

When using LIST for K8s <=1.30, set resourceVersion=0 to read data from the API Server cache, avoiding a full request that hits etcd. To read a large amount of data from etcd, use pagination based on limits.

Effect

Increase access speed and reduce pressure on the control plane.

Prioritize using the Informer mechanism

Suggestion

In large-scale scenarios, frequent LIST requests for a large number of resources can cause significant pressure on controlplane APIServer and etcd. Components that perform frequent LIST operations is suggested to switch to using Informer mechanism.

Effect

The LIST+WATCH mechanism based on Informer can elegantly access the controlplane, improve access speed, and reduces pressure on the controlplane.

Serialization encoding protocol

Suggestion

Use ProtoBuf as API serialization protocol for non-CRD resources firstly.

Effect

Compared with JSON, more efficient in transmission bandwidth, less cost in serialization.

Control Frequency to Access Resources

Suggestion

Client-side frequency control to access large-scale resources.

Effect

Reduce the resource and bandwidth pressure on controlplane.


KubeCon
CloudNativeCon
THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
AI_dev
Open Source GenAI & ML Summit
China 2024
Q & A
Thanks !