UNIVERSITY OF FLORENCE

Department of Information Engineering

Master Degree in Information Engineering

Project report of Quantitative Evaluation of Stocastic Models and Software Architecture and Metodologies

Predictive autoscaling using stocastic models for Kubernetes

Lecturer:

Prof. Vicario Enrico

Students:

Beragnoli Jacopo

Lucchesi Carlo

Orsucci Giacomo

Academic Year 2024/2025

Contents

1 Introduction **3**
1.1 Statement . . . 3
1.2 Project goal and working environment . . . 3

2 System design **4**
2.1 C4 model . . . 4
2.2 Level 1 - Software Systems . . . 4
2.3 Level 2 - Containers . . . 5
2.4 Level 3 - Components . . . 5
2.4.1 System Under Test (SUT) . . . 5
2.4.2 Custom Services . . . 6
2.4.3 Operations and Monitoring . . . 6
2.4.4 Role-Based Access Control . . . 7
2.5 Level 4 - Code . . . 9

3 Experiments **10**
3.1 Machine Specification . . . 10
3.2 Experiments Description . . . 10
3.3 Experiments guide . . . 10
3.3.1 Experiment personalization . . . 11
3.3.2 Setup Details . . . 12
3.4 Arrival Modeling . . . 12
3.5 Rejection Rate Calculation . . . 12
3.6 Experiments Setup . . . 13
3.7 Results . . . 14
3.7.1 Immediate Apply of Recommendations . . . 14
3.7.2 Sliding Window . . . 16
3.7.3 Comments . . . 18

4 Conclusions **19**

A Cost Calculation and Lag Cost Definition **20**

1 Introduction

1.1 Statement

We want to build a micro-service oriented architecture using Kubernetes (K8s) to deploy and scale horizontally our containers (pods in K8s terminology). The goal of the project is to create a complete platform to study the behaviors and the increased capacities of an horizontal scaling based on a message-queue rejection rate. In doing so, technologies such as K6, RabbitMQ, Kafka, Docker, Grafana, Prometheus and Sirio Framework will be used. In detail, we want to model different types of load (Poisson, Uniform, Erlang inter arrival times) with different rates to generate messages. Using arrival information, we utilise a stochastic model of the system to recommend horizontal scaling of service pods in order to meet a rejection rate target.

1.2 Project goal and working environment

The aim of the project is to verify the feasibility and effectiveness of a horizontal autoscaler based on the stochastic properties of the model representing a queueing system. Using the framework Sirio, it is possible to define Stochastic Time Petri Nets that models the behavior of a microservices-based system, evaluating some quantity of interest (like Rejection Rate). The objective is to scale the number of services in the most efficient way while respecting a Service Level Agreement (SLA) and/or Service Level Objectives (SLO), identified in this case as a maximum rejection rate tolerated. The working environment is developed using Kubernetes and consists of the following main components:

- **Arrival Service**: this service generates the requests following a particular stochastic distribution. This is implemented using K6 tool.
- **K6**: a framework used to load test systems.
- **Queue**: this element buffers the requests, allowing the worker to elaborate them in a second moment. We decided to use RabbitMQ for the main queue thanks to the provided possibility of setting the queue size explicitly.
- **Process Service:** this service consumes messages from the queue and processes them. We did a custom implementation in Pyhton that simply executes a busy wait.
- **Monitoring/Visualization:** it's done using Prometheus as metric server, in combination with Grafana for the dashboards.
- **Sirio Controller:** this Java service uses the arrival process CDF (calculated using another custom service) to create an STPN and evaluate the rejection rate. It then recommends to Kubernetes the minimum number of active replicas required to meet the SLA.

2 System design

2.1 C4 model

This project adopts a structured approach to software architecture documentation based on the C4 model: a hierarchical visualization framework that enables clear communication of complex system architectures. The documentation is organized through four distinct levels of abstraction: software systems, containers, components, and code, each serving specific communication needs and technical depth requirements. The C4 model provides a lean graphical notation technique for modeling software system architectures through structural decomposition.

- **Level 1 - Software Systems:** The highest level of abstraction describing systems that deliver value to users, whether human or automated. A software system represents something a single development team builds, owns, and has responsibility for, typically corresponding to team boundaries and deployment units.
- **Level 2 - Containers:** Runtime applications or data stores that must be running for the overall system to function. These include services, APIs, and message queues - essentially the major building blocks that distribute system responsibilities.
- **Level 3 - Components:** Internal elements within each container, showing how components interact with each other and external systems. This level reveals the logical groupings and interfaces that compose each container's functionality.
- **Level 4 - Code:** The most detailed view showing classes, interfaces, and code-level relationships.

This chapter will present and describe the C4 diagrams that have been developed.

2.2 Level 1 - Software Systems

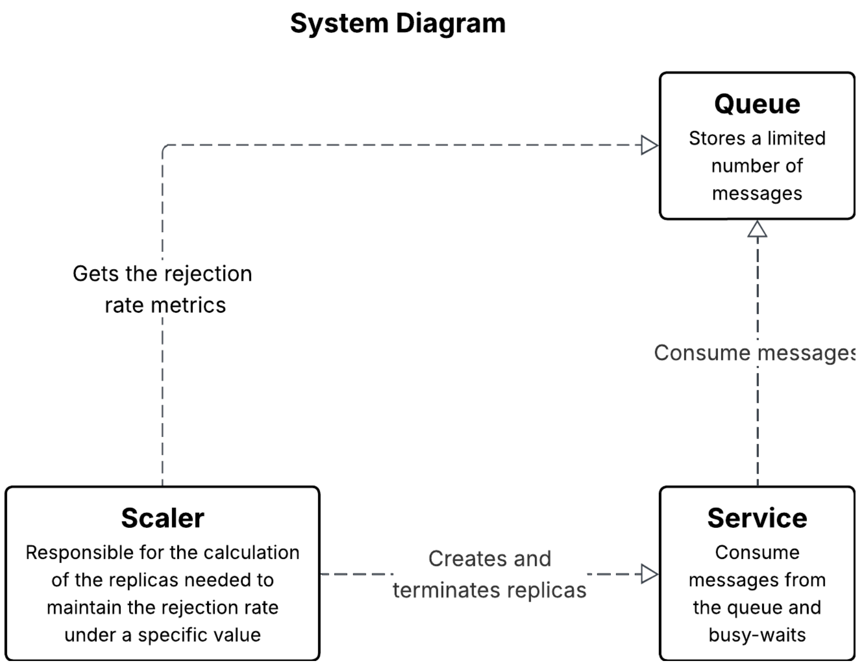

System Diagram
Queue
Stores a limited number of messages
Gets the rejection rate metrics
Consume messages
Scaler
Responsible for the calculation of the replicas needed to maintain the rejection rate under a specific value
Creates and terminates replicas
Service
Consume messages from the queue and busy-waits

Figure 1: System Diagram

As shown in figure 1, the level 1 represents the highest level of abstraction and it highlights its major components and functionalities. Its core comprehends a queue (reached by the messages generated) from which are exposed the necessary metrics to scale our service in order to achieve a target SLA.

2.3 Level 2 - Containers

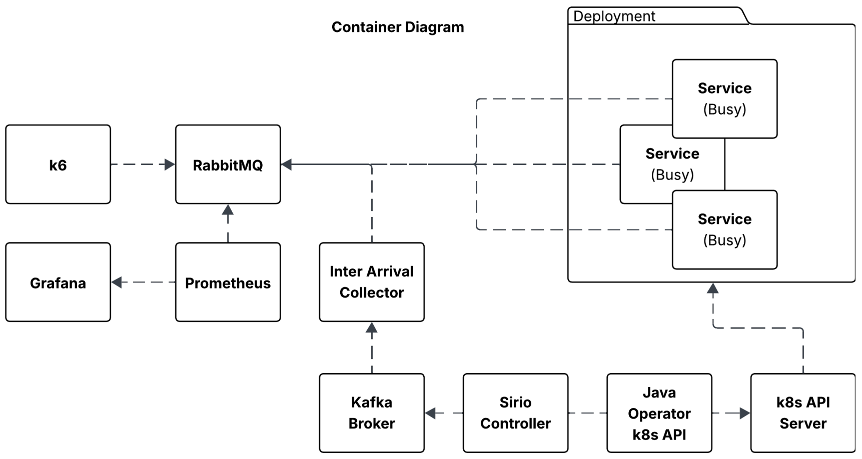


Figure 2: Container Diagram

Going more in depth, as shown in figure 2 we see the container organization, which allows us to detail all the micro services built and their functions. Starting from left we have k6 to generate the load (in our case modeled as a state machine). The messages are stored in a RabbitMQ queue, which exposes the metrics of our interest on Grafana via Prometheus. The Inter Arrival Collector exposes the CDF messages (calculated on the inter-arrivals from the queue) on a Kafka broker that passes the messages to the Sirio Controller component. It uses the CDFs to estimate via the internal model the rejection rate in function of service pods replicas. The recommendation is estimated to keep the rejection rate under the target fixed. On the basis of this estimate, the Kubernetes API is used to scale the services horizontally (their number is increased or decreased).

2.4 Level 3 - Components

At the component architecture level, we have implemented a series of containers, entirely contained within the `oris-predictive-autoscaler` namespace (ns). The architecture is divided into three main logical areas:

1. The **System Under Test (SUT)**, which includes the custom applications and the basic messaging infrastructure (Kafka and RabbitMQ).
2. The **Accessory Monitoring Services**, which provide observability and workload analysis capabilities.
3. The **Security and Access Control (RBAC)** configuration, which governs the permissions of the various components within the cluster.

2.4.1 System Under Test (SUT)

The SUT represents the core of the system and includes both the custom-developed application components (yellow background) and the infrastructure services they rely on (white background). Asynchronous communication and data flow management are handled by two message brokering systems:

- **RabbitMQ**: The message broker, used as the incoming request queue for the service. It is also implemented as a **StatefulSet (sts)** and configured via two **ConfigMaps (cm)**: `rabbitmq-definitions` and `rabbitmq-config`, which define its policies, queues, and initial settings. A **Service (svc)** ensures its accessibility.

- **Kafka**: Used as a notification and data exchange system between the `inter-arrival-collector` and the `sirio-controller`. Being a stateful system, it is implemented as a **StatefulSet (sts)** to ensure stable network identities and persistent storage, managed via a **PersistentVolumeClaim (pvc)**. A **Service (svc)** exposes Kafka within the cluster, allowing components to communicate with it.

2.4.2 Custom Services

These are the components where the specific business logic of the autoscaling system resides.

- **sirio-controller**: Contains the `sirio-controller` **Deployment**, which represents the brain of the system. This controller implements the predictive autoscaling logic, monitoring metrics and making decisions on how and when to scale the service. The deployment's code diagram will be presented in the next chapter, 2.5.

- **inter-arrival-collector**: This **Deployment** is responsible for collecting the inter-arrival times of requests. Acquiring this data is fundamental for the predictive model. Once collected, the data is sent to Kafka to be processed by the `sirio-controller`.

- **python-service**: This **Deployment** represents the target application, i.e. the service that is monitored and whose number of replicas is dynamically managed by the `sirio-controller`.

2.4.3 Operations and Monitoring

These services (green background) are not part of the main application logic (SUT) but are crucial for collecting metrics, analyzing performance, and visualizing the system's state.

- **Prometheus**: It is the central monitoring and alerting system. Implemented as a **Deployment**, it uses a **ConfigMap** (`prometheus-config`) to define the targets from which to scrape metrics (such as `kube-state-metrics` and the SUT applications). A **Service** exposes its interface so that it can be easily reached by Grafana.

- **kube-state-metrics**: An essential service that connects to the Kubernetes API Server to generate metrics about the state of cluster objects (in this case, the number of pods in a deployment). These metrics are then collected by Prometheus to be displayed on Grafana.

- **Grafana**: A visualization tool that connects to Prometheus as a datasource. Its **Deployment** is configured via two **ConfigMaps**: `grafana-datasource` to set up the connection to Prometheus and `grafana-dashboards` to preload custom dashboards.

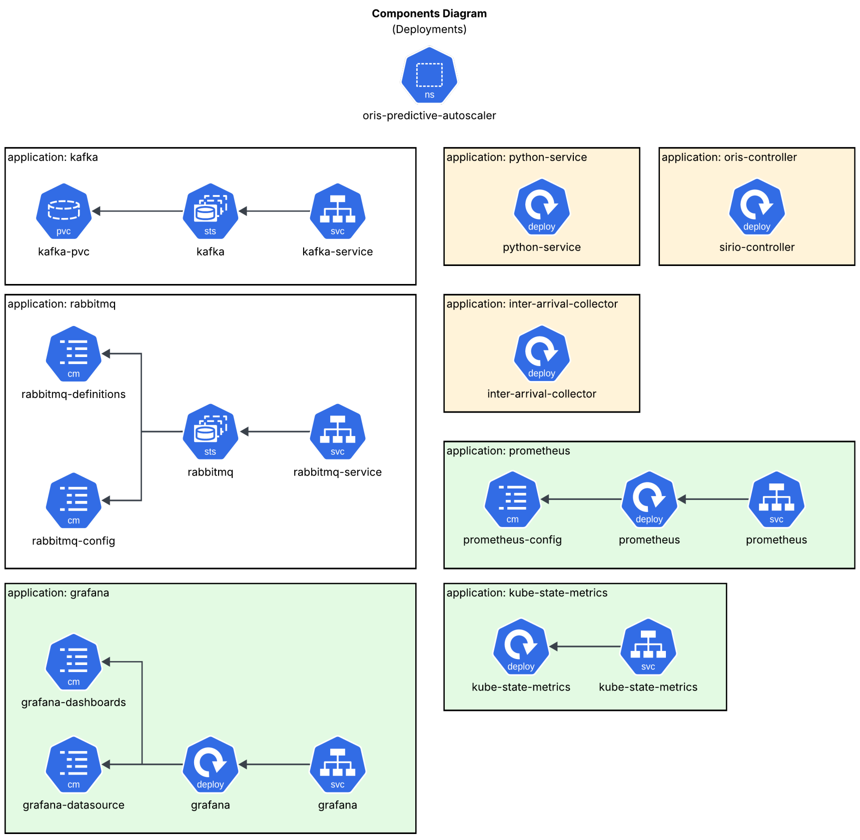


Figure 3: Deployments Diagram

2.4.4 Role-Based Access Control

The second diagram illustrates the RBAC (Role-Based Access Control) configurations that ensure each component operates on the principle of least privilege, accessing only the resources that are strictly necessary.

- **kube-state-metrics-rbac**: In order to generate metrics on the entire cluster, this service needs read permissions on all Kubernetes objects. Its configuration includes:
  - A **ServiceAccount (sa)**: `kube-state-metrics`, the identity used by the pod.
  - A **ClusterRole (c.role)**: Defines `get`, `list`, `watch` permissions at the cluster level.
  - A **ClusterRoleBinding (crb)**: Binds the `ClusterRole` to the `ServiceAccount`, making the permissions effective.
- **sirio-controller-rbac**: Being the heart of the autoscaling logic, the controller must be able to monitor and modify the state of other objects (like Deployments). In this case as well, the configuration uses a **ServiceAccount**, a **ClusterRole**, and a **ClusterRoleBinding** to grant it the necessary permissions to operate at the cluster scale.
- **prometheus-operator-rbac**: This is the most complex configuration because it deals with defining the functionalities and permissions of the Prometheus Operator. The operator requires permissions at both the namespace and cluster levels:

- A **Role** and a **RoleBinding (rb)**: grant it specific permissions within the `oris-predictive-autoscaler` namespace.
- A **ClusterRole** and a **ClusterRoleBinding (crb)**: grant it broader permissions, necessary to discover services to monitor across the entire cluster or to manage resources globally.

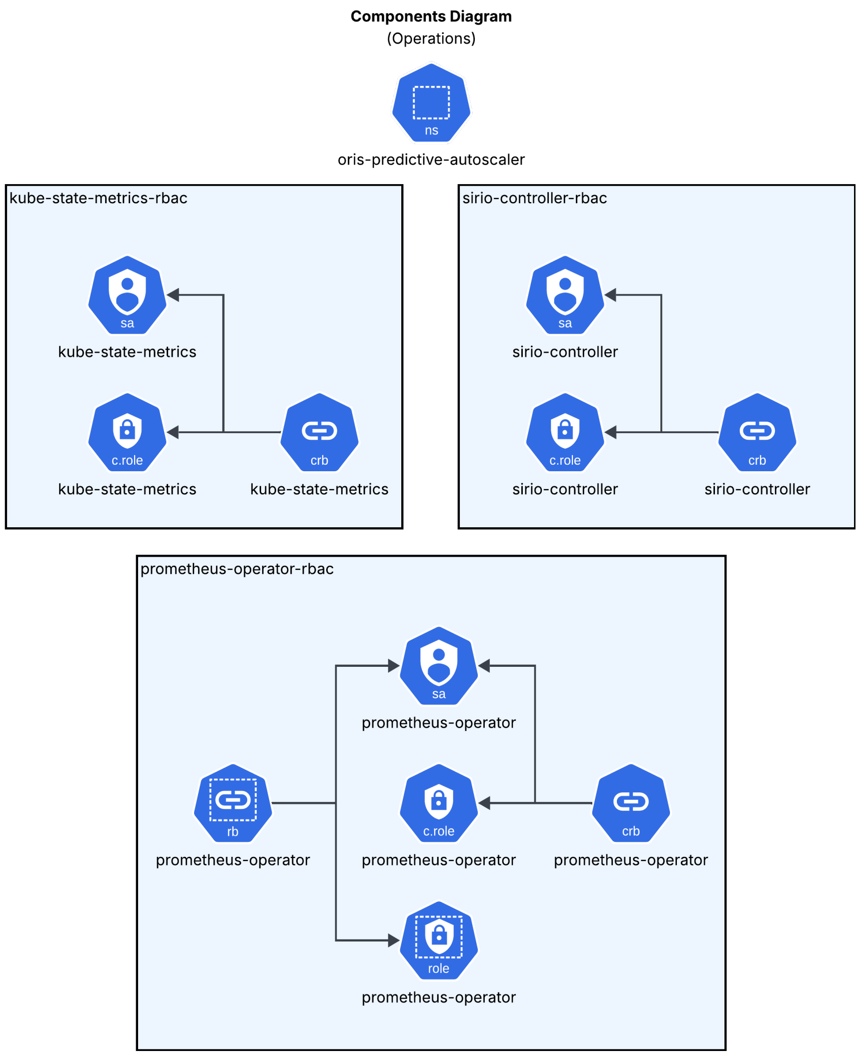


Figure 4: Operations Diagram

2.5 Level 4 - Code

And finally, the code:

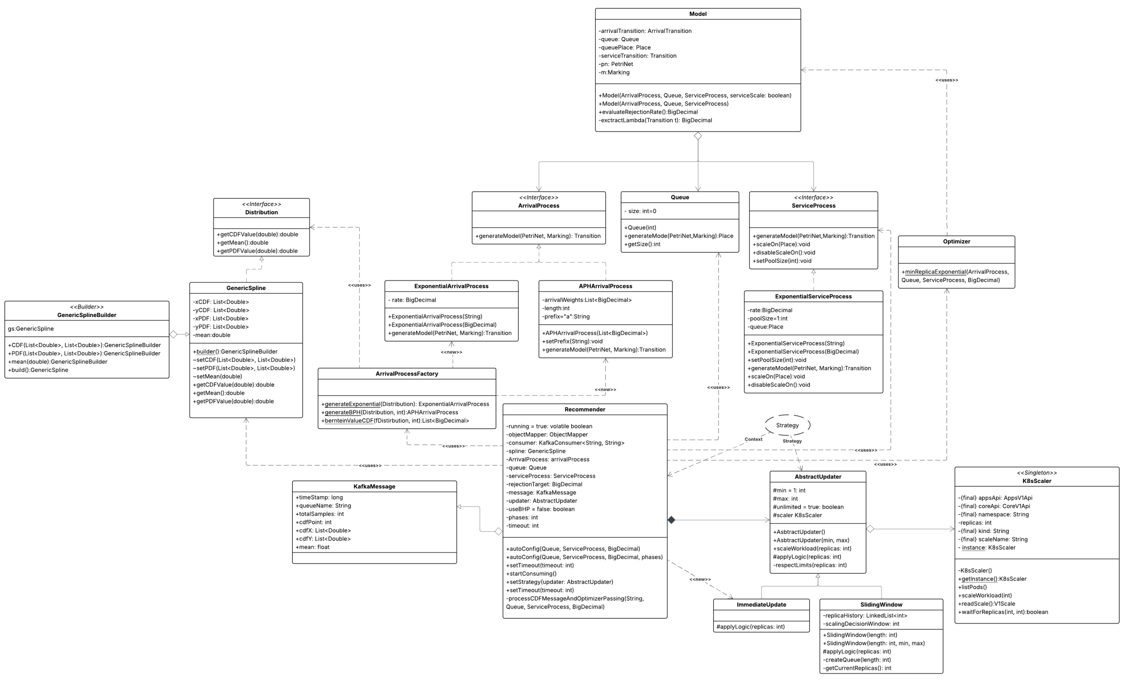

Model
-arrivalTransition: ArrivalTransition
-queue: Queue
-queuePlace: Place
-serviceTransition: Transition
-pn: PetriNet
-m:Marking
+Model(ArrivalProcess, Queue, ServiceProcess, serviceScale: boolean)
+Model(ArrivalProcess, Queue, ServiceProcess)
+evaluateRejectionRate():BigDecimal
-exctractLambda(Transition t): BigDecimal
<<uses>>
<<Interface>>
Distribution
+getCDFValue(double):double
+getMean():double
+getPDFValue(double):double
<<Interface>>
ArrivalProcess
+generateModel(PetriNet, Marking): Transition
Queue
- size: int=0
+Queue(int)
+generateMode(PetriNet,Marking):Place
+getSize():int
<<Interface>>
ServiceProcess
+generateModel(PetriNet,Marking):Transition
+scaleOn(Place):void
+disableScaleOn():void
+setPoolSize(int):void
Optimizer
+minReplicaExponential(ArrivalProcess, Queue, ServiceProcess, BigDecimal)
<<Builder>>
GenericSplineBuilder
gs:GenericSpline
+CDF(List<Double>, List<Double>):GenericSplineBuilder
+PDF(List<Double>, List<Double>):GenericSplineBuilder
+mean(double):GenericSplineBuilder
+build():GenericSpline
GenericSpline
-xCDF: List<Double>
-yCDF: List<Double>
-xPDF: List<Double>
-yPDF: List<Double>
-mean:double
+builder():GenericSplineBuilder
~setCDF(List<Double>, List<Double>)
~setPDF(List<Double>, List<Double>)
~setMean(double)
+getCDFValue(double):double
+getMean():double
+getPDFValue(double):double
<<uses>>
ExponentialArrivalProcess
- rate: BigDecimal
+ExponentialArrivalProcess(String)
+ExponentialArrivalProcess(BigDecimal)
+generateModel(PetriNet, Marking):Transition
APHArrivalProcess
-arrivalWeights:List<BigDecimal>
-length:int
-prefix="a":String
+APHArrivalProcess(List<BigDecimal>)
+setPrefix(String):void
+generateModel(PetriNet, Marking):Transition
ExponentialServiceProcess
-rate:BigDecimal
-poolSize=1:int
-queue:Place
+ExponentialServiceProcess(String)
+ExponentialServiceProcess(BigDecimal)
+setPoolSize(int):void
+generateModel(PetriNet, Marking):Transition
+scaleOn(Place):void
+disableScaleOn():void
<<new>>
ArrivalProcessFactory
+generateExponential(Distribution): ExponentialArrivalProcess
+generateBPH(Distribution, int):APHArrivalProcess
+bernteinValueCDF(fDistirbution, int):List<BigDecimal>
<<new>>
<<uses>>
Recommender
-running = true: volatile boolean
-objectMapper: ObjectMapper
-consumer: KafkaConsumer<String, String>
-spline: GenericSpline
-ArrivalProcess: arrivalProcess
-queue: Queue
-serviceProcess: ServiceProcess
-rejectionTarget: BigDecimal
-message: KafkaMessage
-updater: AbstractUpdater
-useBHP = false: boolean
-phases: int
-timeout: int
+autoConfig(Queue, ServiceProcess, BigDecimal)
+autoConfig(Queue, ServiceProcess, BigDecimal, phases)
+setTimeout(timeout: int)
+startConsuming()
+setStrategy(updater: AbstractUpdater)
+setTimeout(timeout: int)
-processCDFMessageAndOptimizerPassing(String, Queue, ServiceProcess, BigDecimal)
<<uses>>
<<uses>>
Strategy
Context
Strategy
<<uses>>
<<uses>>
KafkaMessage
+timeStamp: long
+queueName: String
+totalSamples: int
+cdfPoint: int
+cdfX: List<Double>
+cdfY: List<Double>
+mean: float
AbstractUpdater
#min = 1: int
#max: int
#unlimited = true: boolean
#scaler K8sScaler
+AsbtractUpdater()
+AsbtractUpdater(min, max)
+scaleWorkload(replicas: int)
#applyLogic(replicas: int)
-respectLimits(replicas: int)
<<Singleton>>
K8sScaler
-{final} appsApi: AppsV1Api
-{final} coreApi: CoreV1Api
-{final} namespace: String
-replicas: int
-{final} kind: String
-{final} scaleName: String
- instance: K8sScaler
-K8sScaler()
+getInstance():K8sScaler
+listPods()
+scaleWorkload(int)
+readScale():V1Scale
+waitForReplicas(int, int):boolean
<<new>>
ImmediateUpdate
#applyLogic(replicas: int)
SlidingWindow
-replicaHistory: LinkedList<int>
-scalingDecisionWindow: int
+SlidingWindow(length: int)
+SlidingWindow(length: int, min, max)
#applyLogic(replicas: int)
-createQueue(length: int)
-getCurrentReplicas(): int

Figure 5: UML Class Diagram

The UML class diagram for the Sirio controller is shown in figure 5. It's important to notice the part fo the code that, recovered the CDF approximation from Kafka, uses it to create the stochastic model of the system. This logic is encapsulated inside the KafkaMessage class. The logic of optimization that finda the optimal number of replicas is inside the Optimizer class. Follow this link for a better visualization.

3 Experiments

In this section we will discuss the structure of the test executed and their result.

3.1 Machine Specification

We executed the following tests using this environment:

- **Machine**:
  - Type: HP OmniBook Ultra Flip 14
  - CPU: Intel® Core™ Ultra 7 - 258V (4.8 GHz, 12 MB cache L3, 8 core, 8 thread)
  - RAM: 32 GB LPDDR5x-8533 MT/s
- **Software**:
  - OS: Ubuntu 24.04.3 LTS (Noble Numbat) x86_64
  - Host: Windows Subsystem for Linux - Ubuntu (2.5.7.0)
  - Kubernetes: minikube version: v1.36.0, configured with 4 cores and 16GB of RAM

3.2 Experiments Description

The experiments have the goal of emulating different types of traffic, and showing the system behavior. The possible experiments can be performed choosing a distribution, and then specifying the set of parameters for every state of a state machine. In other words, if the user selects 4 different rates for the distribution selected the load is composed of 4 circularly executed states with the rates specified. The duration of each state is editable from the code.

For example, choosing a uniform distribution with all rates value such that min=max (to model the deterministic arrival process), selecting a test duration of 9 minutes with rates = ([1,1];[2,2];[3,3];[5,5]) the result is an arrival process that generates a load with 1 message/s, than 2 messages/s, 3 messages/s and 5 messages/s for 1 minute each. At this point 4 minutes has been passed so the arrival process restarts circularly from 1 message/s and so on until the end of the test.

3.3 Experiments guide

In this section we include a miniguide to reproduce our experiments. First of all it is necessary to clone the GitHub repository at this link (https://github.com/Berags/oris-predictive-autoscaler). To start the project is necessary, as a minimum dependency, to have installed Minikube. After having installed successfully Minikube, execute the following command:

```
minikube start
```

then, it is sufficient to position the terminal on the directory of the project where it is present the .sh file named ./start.sh and execute it with:

```
./start.sh
```

This contains all the instruction necessary to build and run our project (i.e. all the containers and components). Don't mind if it takes a while, it is perfectly normal due to the project complexity. At this point, as reported in «port-forward.sh» you can monitor:

- The RabbitMQ queue at «http://localhost:15672»
- Prometheus at «http://localhost:9090»
- Grafana (with all the dashboards of interest shown below) at «http://localhost:3000», the default username and password is `admin`.
- Kafdrop (for the Kafka queue) at «http://localhost:9000»
- It's possible to monitor the state of the Sirio Controller's JVM using *jconsole* with the address «http://localhost:9000».

At this point everything is ready to start the tests. In order to do it, go in the folder of «k6 »and execute:

```
./build-and-run.sh
```

This command will show the following menu:

```
1. Exponential
2. Poisson (λ<100)
3. Uniform (Use min = max for deterministic)
4. Erlang (k, λ)
5. Hyper Erlang (k, λ, weight)
6. Exit
----------------------------
Insert your choice:
```

Here you can choose the distribution of the load you want to simulate. Then, remains only to choose the requested parameters as described before and how you can see below. Finally, the test starts and the console reports some logs. To monitor the entire system consult the addresses above.

3.3.1 Experiment personalization

Copying the repository you can essentially modify the code as you prefer but if you want to modify the transition time is sufficient to modify the following instruction:

```
const transitionTime = DistributionFactory.getDeterministic(60);
```

in rabbitmq-test.js and with «60» that are the seconds of permanence in each state. If you are not interested in a deterministic transition you can substitute it, for example, with an Exponential one with:

```
const transitionTime = DistributionFactory.getExponential(20);
```

Please notice how we modeled arrival processes of our interest using the js library included in the code repo under k6/lib (all the documentation inside), but eventually you can use also the others distributions supported.

3.3.2 Setup Details

Here are reported more details on the usage of the *.sh* file inside the main directory. Essentially, the fundamental ones are «start.sh» and «build-and-run.sh» but since ./start.sh does a lot of work and when something goes wrong is not everytime necessary to rebuild everything, so we provided «port-forward.sh» for the port forwarding only and «delete_and_deploy_sirio.sh» to, as its name suggests, delete and redeploy only sirio. The «delete_and_cdf_genereator.sh»does the same, but for the *inter-arrival-collector* service.

3.4 Arrival Modeling

For all the test, we modeled in Sirio the Arrival process as an Exponential with parameter λ equal to the inverse of the average inter-arrival time of the messages in the queue (or λ equals to the average messages per second, that is equivalent).

We also tried using Bernstein phase types, but we encountered a problem. It seems that the Sirio model of a Bernstein phase type uses a rapidly increasing amount of RAM in the increasing number of phases. For instance, we tried 50 phases and the scaler tried to instance in the heap more than 768MB, making the pods memory overflow and be killed by Kubernetes. With a lower number of phases, we didn't achieve a good enough approximation to be used in the scaler. In the other hand, trying to extend the amount of RAM available to the Sirio pod make the machine's OS trashing most of the time.

In addition to that, we found the approximation with the exponential more that capable to a uniform arrival rate. Then, having a microservice controller that consumes so much resource seem odd and counterproductive, and against the principles of microservices itself.

3.5 Rejection Rate Calculation

Here we want to discuss how we calculated the rejection rate, and the reasons for it. For our experiments, we wanted to calculate a relative rejection rate. In fact, we consider this approach more robust if compared to an absolute target.

The rejection rate, at least in terms of expected value, can be defined as: *the probability that a packed get rejected by the queue.* To verify this condition, two pre-conditions must be true:

- The Queue is full.
- The arrival process is habilitated to push a new message.
- The arrival process extracts a time to fire smaller than the service process.

Considering a stochastic model expressed in term of a Stochastic Time Petri Net (STPN), we are interested in only three elements: the place of the queue, the last transition of the arrival process that goes into the queue, and the first transition of the service that pulls from the queue (most of the time the service is represented with only a transition). For this problem we assume that the stochastic properties of the model depends only on the marking state. So, let R be the event of a rejection, $Q = q_{max}$ the event of a marking with the queue full, A the event of a marking where the last arrival transition has a token as a precondition, t_a and t_s the values sampled from the arrival and service transitions, from the previous conditions we can derive the equation 1:

$$\begin{aligned} P(R) &= P(Q = q_{max} \land A \land t_a < t_s) \\ &= P(t_a < t_s | Q = q_{max} \land A) P(Q = q_{max} \land A) \end{aligned} \tag{1}$$

Considering that we used a Markovian model, the arrival and service transitions samples their times to fire form exponential transitions. So, let λ_a and λ_s of two transition when the queue is full and the arrival process able to push a new token into it, we can rewrite equation 1 as 2:

$$\begin{aligned} P(R) &= P(t_a < t_s | Q = q_{max} \wedge A) P(Q = q_{max} \wedge A) \\ &= P(Exp(\lambda_a) < Exp(\lambda_s)) P(Q = q_{max} \wedge A) \end{aligned} \tag{2}$$

Can be easily derived the probability of an exponential samples before another as in equation 3, so that the final rejection rate formula is 4.

$$P(Exp(\lambda_a) < Exp(\lambda_s)) = \frac{\lambda_a}{\lambda_a + \lambda_s} \tag{3}$$

$$P(R) = \frac{\lambda_a}{\lambda_a + \lambda_s} P(Q = q_{max} \wedge A) \tag{4}$$

Considering that given a closed model Sirio is capable to enumerate all the reachable states, using the steady state analysis we can get the probability of having a state where the queue is full and the arrival transition is enable, while using the *MarkingExpresions* getting transition rates is trivial.

Here below, to highlight how this formula can be translated in practice, is reported the code that calculated the rejection rate.

```java
BigDecimal rejection = BigDecimal.ZERO;
Map<Marking, BigDecimal> results = RegSteadyState.builder().build().compute
    (pn, m).getSteadyState();
for (Marking tmp : results.keySet()) {
     if (tmp.getTokens(queuePlace) == queue.getSize() && pn.isEnabled(
    arrivalTransition, tmp)) {
         BigDecimal currentRejection = results.get(tmp);

         BigDecimal arrivalRate = extractLambda(arrivalTransition, tmp).
    setScale(8, RoundingMode.HALF_UP);
         BigDecimal serviceRate = extractLambda(serviceTransition, tmp).
    setScale(8, RoundingMode.HALF_UP);

         currentRejection = currentRejection.multiply(arrivalRate).divide(
    arrivalRate.add(serviceRate), 8, RoundingMode.HALF_DOWN);

         rejection = rejection.add(currentRejection);
     }
}
return rejection;
```

3.6 Experiments Setup

In all tests we set some common parameters:

1. Rejection Rate: 5%.
2. Service Rate: 1.
3. Duration of states in the arrival process: 60s.
4. Total test duration: 20 minutes.

For the experiments with constant arrival time, we defined a state machine with the following parameters: [0.33,0.33],[0.2,0.2],[0.11,0.11],[1,1],[0.14,0.14]. In this, for every state we sample the same inter arrival time with the following expected messages per second: 3, 5, 9, 1 and 7. Considering that the state machine is cyclic, this behavior repeats periodically.

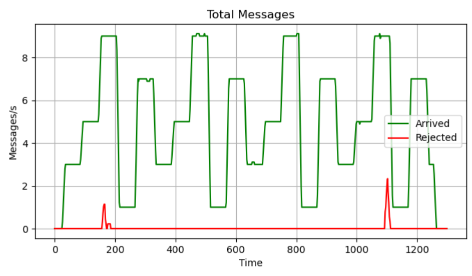
Then, we tried a stochastic workload using exponential as inter arrival time distributions in the machine states. The parameters used are: 3, 5, 7, 1, 9. These are also the expected messages per second of the respectively state.

In this report, we separated the logic of establishing the optimal number of replicas (the Recommender) from the one that communicates with Kubernetes (Updater). This allows to apply different scaling logics to the workers. In the following we will confront an immediate application of the recommendation, versus a sliding window approach that waits a series of downscale recommendation before applying it.

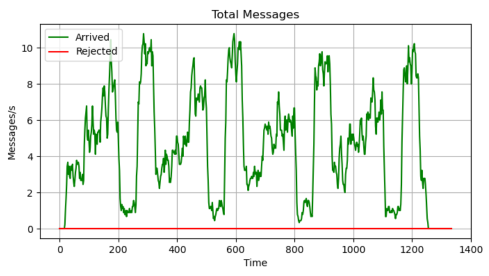
3.7 Results

3.7.1 Immediate Apply of Recommendations

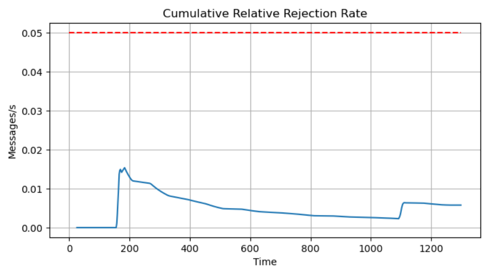
In executing this experiment, both for constant and exponential arrival rates, we measure the arrived and rejected messages in figures 6a and 6b.



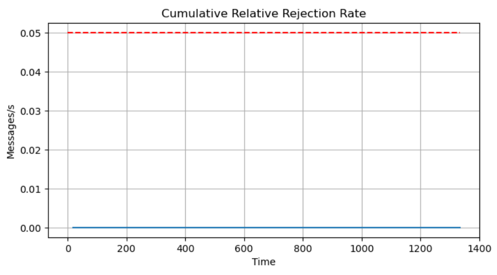
(a) Arrived and Rejected messages per second in constant rates.



(b) Arrived and Rejected messages per second with exponential arrival times.



(c) Cumulative Rejection rate for constant arrival time experiment.

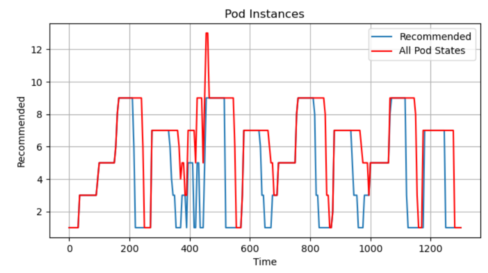


(d) Cumulative Rejection rate for exponential arrival time experiment.

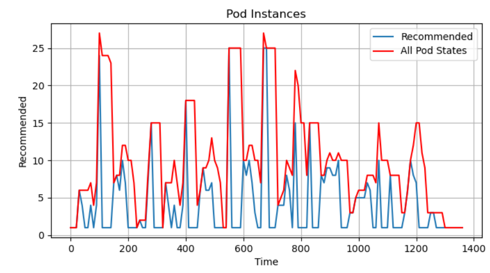
Figure 6: Results of tests with immediate update.

As said, the goal of the control is to keep a rejection rate below 5%. For both of them, rejection rate was under the threshold, with even the exponential without rejected messages at all.

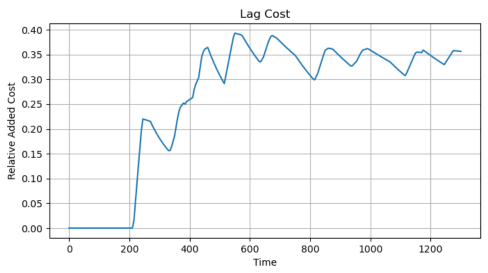
We wanted to see not only if the target was achieved, but also the evolution in the number of recommended pods. The trend of the scaling can be seen in figures 7a and 7b, with the respective lag cost in figures 7c and 7d. More detail on how the costs are calculated can be found in section A.



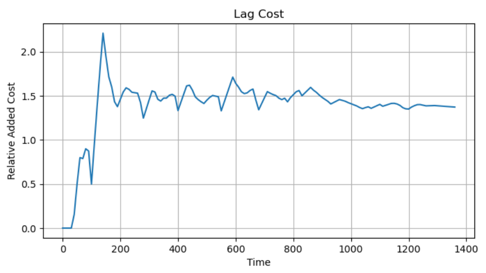
(a) Number of pods trend for constant arrival time experiment.



(b) Number of pods trend for exponential arrival time experiment.



(c) Cumulative lag cost for constant arrival time experiment.



(d) Cumulative lag cost for exponential arrival time experiment.

Figure 7: Measure of pod instances with immediate update.

The surge of lag cost came mainly from downscaling. In fact, when the Sirio Scaler recommends to scale up the pods Kubernetes immediately starts creating them to match the new incoming traffic. On the other hand, when Sirio recommends a downscale the surplus pods cannot be terminated instantaneously, mainly because they are still elaborating some elements.

Another issue that makes lag cost spike is that pods can be much higher than the recommendation. This happens when recommendation drops and rises very shortly. In fact, the first drop makes most of the pods terminating, but when the second rise comes, given that terminating pods cannot be recovered, we need to instantiate even more to compensate. In this way the system satisfies the recommendation very fastly, but we still have all the old pods terminating, so the total number of pods still alive is, possibly, much higher than recommended. Please, to note this behavior observe the figure 7.a around time 400s. It's intuitive how this phenomenon leads to a high lag cost; this is why we decided to explore a sliding windows recommendation policy.

A summary of this results can be seen in table 1.

	Constant	Exponential
Recommended Cost	6030	2880
Effective Cost	8180	6835
Relative Lag	35.66%	137.33%
Rejection Rate	0.58%	0.00%

Table 1: Summary of the results for the immediate application of recommendations.

3.7.2 Sliding Window

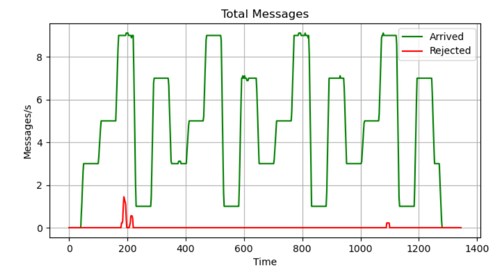
In our experimenting we also tried to implement an updater that uses a sliding window to reduce the lag cost. In the case of a upscaling recommendation, the system will immediately implement it, in the case of a downscaling the system maintains some inertia before doing it (i.e. the system waits a certain number of requests before doing the downscaling).

More precisely, for every downscaling the recommended value is inserted into a history. When the history reaches a certain dimension, and it has values smaller of the current value, the target is updated with the maximum in the history. Every time the target is changed the history is cleared.

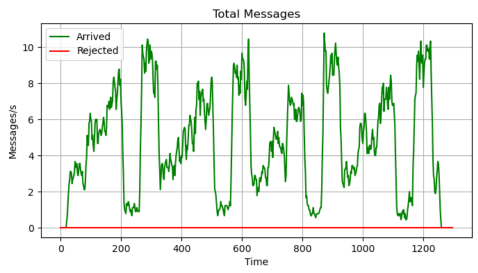
For more details, below there is the snippet of code that implements this behavior.

```
int currentReplicas = getCurrentReplicas();

if (newReplicas > currentReplicas) {
  replicaHistory.clear();
  return newReplicas;
} else if (newReplicas < currentReplicas) {
  replicaHistory.add(newReplicas);
  if (replicaHistory.size() > scalingDecisionWindow) {
    replicaHistory.removeFirst();
  }

  if (replicaHistory.size() >= scalingDecisionWindow
        && replicaHistory.stream().allMatch(r -> r <= currentReplicas)) {
    int maxInHistory = replicaHistory.stream().max(Integer::compareTo).
   orElse(newReplicas);
    replicaHistory.clear();
    return maxInHistory;
  }
} else {
  replicaHistory.add(newReplicas);
}
return currentReplicas;
```

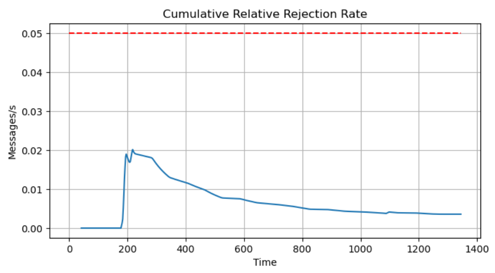
As in the precedent couple of tests, in the figure 8 we report the trend of arrived messages confronted with the cumulative rejection rate, but studying the sliding window implementation.



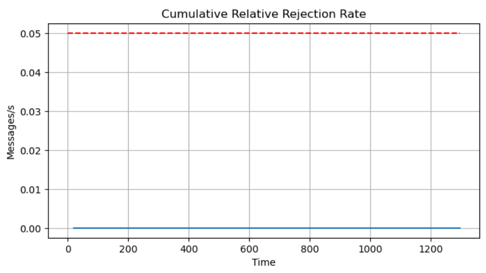
(a) Arrived and Rejected messages per second in constant rates.



(b) Arrived and Rejected messages per second with exponential arrival times.



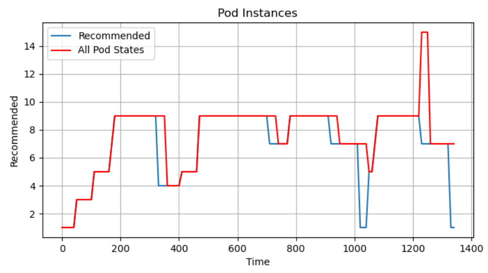
(c) Cumulative Rejection rate for constant arrival time experiment.



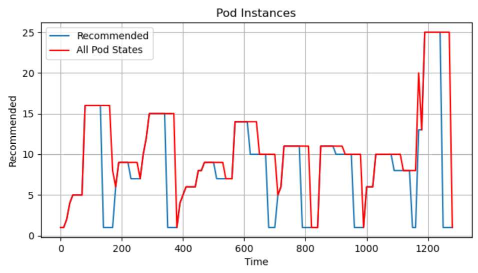
(d) Cumulative Rejection rate for exponential arrival time experiment.

Figure 8: Results of tests using the sliding window.

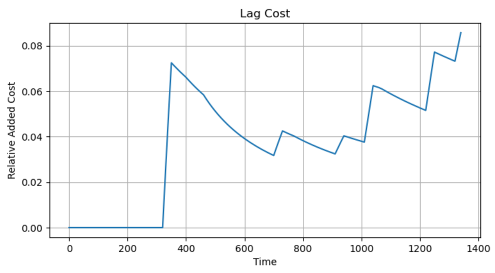
In the same way, the following figures shows the recommended pods with the associated cumulative lag cost with sliding window downscaling policy. It is well highlighted how the number of pods is more stable, avoiding excessive peaks and lags as previously seen.



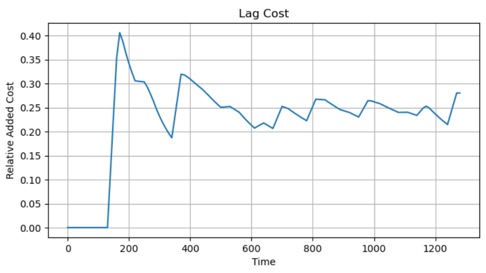
(a) Number of pods trend for constant arrival time experiment.



(b) Number of pods trend for exponential arrival time experiment.



(c) Cumulative lag cost for constant arrival time experiment.



(d) Cumulative lag cost for exponential arrival time experiment.

Figure 9: Measure of pod instances using the sliding window.

Lastly, table 2 report a summary of experiments using the sliding window approach.

	Constant	Exponential
Recommended Cost	4725	5420
Effective Cost	5130	6940
Relative Lag	8.57%	28.04%
Rejection Rate	0.36%	0.00%

Table 2: Summary of the results for the sliding window approach.

3.7.3 Comments

From the above result many things can be observed. Before commenting them, it's important to notice that the total amount of messages generated in every case is mostly the same.

The first result that we can notice is that, introducing a sliding window reduces significantly the relative cost introduced by Kubernetes lag. However, in the exponential case this reduction is mostly due to a rising in the recommended cost. In fact, in the case of a constant arrival time we have got a lower total cost, but in the case of the exponential not a significant change (in reality a slight increase, but that can explained by run-to-run variations).

Certainly the sliding window strategy made the number of pods more stable during time, avoiding excessive peak usage of resources. To better frame this concept, in previous experiments in a different setting and a different machine the exponential case without sliding window always crashed, while using it allowed for a termination without issues.

Another interesting trend that can be seen in this data is that, for some reason, the constant inter arrival time seems harder to manage than the exponential one. In both cases the constant arrival rate is characterized by some rejected messages, while the exponential no. For this phenomenon we can't say if it happened by chance of if there is some hidden reason.

Lastly, something can be said in reference to the very low recommended number cost achieved (in theory) in the exponential case without sliding window. We can see how it's very common seeing it going to 1 (the minimum) due to the fact that both the arrival and service processes are stochastic, making the queue going empty. This makes the CDF generator component of the system fail to generate valid CDFs. If Sirio doesn't receive any messages for a given time (10s as our default choice) it simply *thinks* that there is not load, and recommend for the minimum amount of pods (1 in this case). It's important to notice that terminating pods are probably still elaborating at least the last messages that they were able to pull, factually discharging the queue. These two phenomenon combined make possible to achieve very low recommended cost, but still a 0 rejection rate. The effective cost metric catches this behavior.

4 Conclusions

In this experiment, we developed a microservices-based system with the aim of verifying the feasibility of workload scaling using the stochastic properties of the arrival and service processes. We developed a scaling system that models a STPN using the Sirio framework and uses its quantitative evaluation for an informed scaling mechanism (in our case based on the rejection rate).

We have found that, assuming we approximate the arrival process with an exponential form, this is also effective in the case of a deterministic time between arrivals, which is positive considering the ease with which we can analyse Markovian systems. Even considering the impossibility of using Bernstein phase types due to high resource consumption, we believe that the increased accuracy does not compensate for heavier execution.

In fact, we argue that for arrival processes with a high number of messages per second (condition that this type of scaling are design for) the shape is not so relevant. So, it's better to have an easier model to obtain quick result and be more reactive.

Another aspect to consider is that, for such low rejection rates, the system practically prescribes a number of service replications such that the composite service rate corresponds to the arrival rate. Even if this can conclude in favor of an even simpler solution, the real main advantage of a stochastic approach is the flexibility of the system. We strongly advocate for a use of the Sirio Framework oriented to transient analysis instead of steady state. With this goal in mind, it is trivial to think, as a future development, of a scaler that uses the number of elements in the queue to predict the rejection rate over a short period of time. Thus, to further reduce the number of replicas with a low queue, while increasing processing on a nearly full queue.

In conclusion, we have demonstrated the feasibility of horizontal resizing using an informed stochastic model, laying the foundations for further developments, for example based on machine learning.

A Cost Calculation and Lag Cost Definition

Applying the recommended target replicas on Kubernetes can take some time. In the case of incrementing there is the cost time of create a new pod and connecting it to the Kubernetes network, while decreasing the replicas meant a safe shutdown of the pods, assuring a consistent behavior. We wanted to measure the cost added from this time delays, as the difference between the total pods instantiated in every state (Creating, Running, Terminating, ecc.) and the recommended one.

Without loss of generality, we choose to measure cost in terms of $pods \cdot time$, so that the cost of the system is both proportional to the quantity of resources used and the time we use them. For instance, a system that executes 5 pods for 3 seconds has a cost of 15. It's worth noting that this type of billing is common in many types of cloud providers.

So, given any sequence of measured pods $pods_i$ for $i = 1, \dots, t$, with a sampling period of T_s, the total cost of the control can be (approximately) calculated as 5:

$$C = T_s \sum_{i=1}^{t} pods_i \tag{5}$$

So, be C_{rec} the recommended cost obtained by the sequence of pods dictated by the Sirio Scaler, and C_{real} the real cost of all existing pods, we calculated the relative lag cost as in equation 6:

$$I = \frac{C_{real} - C_{rec}}{C_{rec}} = \frac{C_{real}}{C_{rec}} - 1 \tag{6}$$

Giving the formulas in 5 and 6, both can be calculated for a particular time t by truncating the sequence $pods_i$ for $i \leq t$.

Note that for how we defined I, in theory it can be smaller than 1. In practice, we even consider initializing pods for the actual cost, and that makes it's very unlikely that Kubernetes waits so much to start creating them; this eventuality is negligible. However, if needed we can force the numerator to be bigger or at least equal to the denominator in equation 6.

We want to state that this measure of lag doesn't necessary means a bad implementation, but only a measure of the difference between a perfect control and the actual one.