Self-Adaptations in Large Scale Microservices Architecture using Kubernetes Operators

Jie Shen Beh
The University of Adelaide
Adelaide, Australia
jieshen.beh@adelaide.edu.au

Claudia Szabo
The University of Adelaide
Adelaide, Australia
claudia.szabo@adelaide.edu.au

ABSTRACT

Self-adaptive microservices are essential to high availability and automation. Kubernetes has proven to be the most-used container orchestration tool, but the default Kubernetes Horizontal Pod Autoscaler (HPA) has a default immutable formula and limited support of metrics to perform self-adaptation. This paper aims to benchmark the use of Kubernetes Operators against HPA. The statistically significant results has shown that that there is a difference in the mean average HTTP duration in seconds between HPA and Kubernetes Operators.

1 INTRODUCTION

A microservice architecture encompasses a variety of small loosely-coupled computer systems that work together to form a large distributed software.[1] Their counterpart is a monolithic architecture where all software processes are contained in one single system.[2]

The main benefits of microservices are its ease of scaling and polyglot programming. The independence of each microservice allows a greater control in determining which service to scale, as compared to scaling the entire monolithic solution. With technologies such as event buses[3] (a pipeline that receives events) and API gateways[4] (an entrypoint between clients and the backend microservices), software teams are able to build microservices with different programming languages and frameworks.

A self-adaptive system is capable of adapting its own behaviours to respond towards different states that occur during system runtime [12]. In order to trigger a self-adaptation, the system will continuously monitor runtime metrics, decide whether the runtime metrics are aligned according to predefined adaptation goals and potentially carry out certain actions to actualize those goals. Without self-adaptation, the general approach is to implement metrics monitoring and alerts for manual intervention. However, the time difference between the alert and actual intervention will often lead to a breach in Service Level Agreements (SLA)[5] and a loss of business revenue.

Kubernetes[6] is an open-source container orchestration platform that was developed by Google. A container is a standalone piece of software that can be executed reliably across different environments [13]. Docker[7] achieves this by isolating the application code from its system runtime and encapsulating its other software dependencies. Recent reports have shown that containers are abstracting microservices in modern architecture. In the last two years, the average number of containers per organisation has doubled across the analysis of 1.5 billion containers run by tens of thousands of Datadog[8] customers [8].

In Kubernetes, a Pod is known as the smallest deployable component that abstracts containers. Pods are powered by read-only container runtime images such as Docker Images. Container runtime images such as Docker images[9] are created as the result of containerisation and they are now usable by Kubernetes components. In order to ensure a specified number of pods are operational, a Kubernetes ReplicaSet can be deployed. As it forms an interface for most components, Deployments and StatefulSets are powered by ReplicaSets to ease pod deployments. A Kubernetes Service is a component that makes other Kubernetes components accessible. The most common use case is to assign a static cluster-level IP address for Pods, as they are highly disposable during runtime.

[1]https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices
[2]https://aws.amazon.com/microservices/
[3]https://aws.amazon.com/event-driven-architecture/
[4]https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do
[5]https://www.atlassian.com/itsm/service-request-management/slas
[6]https://kubernetes.io/
[7]https://www.docker.com/resources/what-container/
[8]https://www.datadoghq.com/
[9]https://docs.docker.com/get-started/overview/

1.1 Motivation

Horizontal Pod Autoscaler[10] (HPA) is a built-in component that allows automatic pod scaling according to limited runtime metrics such as CPU. However, most autoscalers such as HPA and AWS EC2 Auto Scaling groups[11] are inflexible in providing customisations due to the underlying adaptation strategy. HPA's immutable formula is defined as follows:

$$desiredReplicas = \left\lceil currentReplicas * \frac{currentMetric}{desiredMetric} \right\rceil$$

CPU metrics alone is insufficient to determine the performance of a system. According to Pahl [15], it is often difficult to identify if a system is undergoing a normal behaviour or anamolous behaviour with CPU alone. For example, a system with 300 users may have a lower CPU utilisation than a system with 1000 concurrent users. Additionally, HPA support of custom metrics is highly limited.

This paper benchmarks the performance of Kubernetes Operators that allows the use of custom business metrics and logic against HPA. The remainder of the paper is organised as follows. Section 2 will include a literature review of related work. In Section 3, the paper will explore methodologies used and introduce Kubernetes Operators. Section 4 will demonstrate the experimental setup followed by the results obtained in Section 5. The paper will discuss several limitations in Section 6 and draw conclusions in Section 7.

2 LITERATURE REVIEW

2.1 Machine Learning

Machine learning[12] (ML) utilises applied statistical and computer science knowledge on vast amount of data to imitate how humans learn over time. In self-adaptations, ML inference allows systems to perform proactive adaptations instead of reactive adaptations. Proactive adaptations are able to prevent Quality-of-Service (QoS) problems by forecasting it and making a change before it happens.

Sanctis et al. proposed a self-adaptation microservice architecture that runs on different Internet of Things [13](IoT) to improve QoS [3]. Past data of Edge devices have been leveraged to dynamically adapt the runtime behaviours and maintain a high QoS. By performing lightweight computations on metrics such as battery level, the ML model continuously predicts the expected future QoS for a given time interval. Some adaptations such as reducing the data transfer can be carried out to preserve battery level.

Reinforcement learning (RL) is a technique in machine learning where a ML agent goes through a workflow of interactions in their training environment to learn how to make a specific decision, which is scaling pods in this case [17]. Horovitz and Arian presented a novel algorithm called Q-Threshold to dynamically adjust thresholds without human intervention in runtime [7].

The main challenge in integrating machine learning models to allow self-adaptation is its underlying complex architecture that will compete for crucial resources such as CPU and internet bandwidth. Training and accurate inference incurs high storage and processing resources that conflicts with the constraints in lightweight microservices, affecting SLAs [5]. Rossi argues that most RL algorithm in the research space experience a slow convergence rate during training [14].

2.2 Control Loops

Aderaldo et al. introduced Kubow [1], a practical extensible self-adaptive system that extends beyond Kubernetes and Rainbow. The Rainbow framework consist of multiple engineering mechanisms to gather data and act on a certain adaptation strategy [4] [2]. According to a proportion of Rainbow's original creators, there is a fundamental mismatch between architectural models provided by academics and the needs of industrial organisations that rely heavily on cloud technologies [10]. However, the additional complexity using Rainbow components can be easily replaced by using HPA.

Service meshes utilises a inter-service communication strategy as a platform to securely manage and observe the behaviours of microservices [11]. This strategy falls under the planner category. The technical implementation of service meshes include providing a

[10]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[11]https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-groups.html
[12]https://www.ibm.com/au-en/cloud/learn/machine-learning
[13]https://www.oracle.com/au/internet-of-things/what-is-iot/

sidecar proxy[14] to each microservice to facilitate communication. Some additional use case include A/B testing, canary deployment or seamlessly enforcing security policies.

According to Mendonca and Aderaldo [9], an open source service mesh platform called Istio[15] is the most popular service mesh platform. Istio employs a data plane and a control plane that deploys Envoy sidecar proxies to facilitate network communication and routes traffic according to user-defined rules respectively. However, Mendonca and Aderaldo state that there are currently no service mesh platforms that provides self-adaptation, including Istio [9]. Most service mesh platforms interact with the component responsible for making self-adaptations, for instance the Kubernetes HPA.

2.3 Queuing Theory

Queuing theory involves the the study of mathematical models in congestion among queues. In terms of microservices, queuing theory models the system as a queuing network to predict application performance during runtime. Based on the network system arrival times, it will approximately follow statistical distributions such as M/G/1-PS [16]. Gias et al. developed a Layered Queuing Network to solve the optimisation problem in determining the number of pods needed in a microservices architecture [6]. Although queuing models have been heavily researched by the industry, the main drawback is the centralised model that becomes the bottleneck in large-scale microservices architecture.

3 METHODOLOGY

The methodology used in the process was to benchmark the performance of Kubernetes Operators against HPA using a case study. The contingency plan is to attempt building the executor while understanding the core problems that hinders progress. More details of the prototype implementation will be outlined in the experimental setup.

3.1 Definition of Adaptation Goals And Behaviours

Firstly, adaptation goals were defined according to past data trends or custom business needs. Past data gathered from monitoring tools can serve as a starting point for defining specific goals. A trivial example is CPU, where most web servers would experience an increase in CPU usage over time if there is an increase in user traffic. Additionally, adaptation goals can be derived from business-wide policies to maintain a SLA.

In order to change the state of the system, adaptation actions must be clearly outlined and executed. A potential behavioural adaptation to reduce CPU usage could be increasing the number of web servers to distribute the user traffic equally.

3.2 Proposed Framework

There are four main parts in the proposed framework: Kubernetes, Logging and Monitoring, Observability and Custom Scaler. The Kubernetes cluster will encompass all containerised applications in its own ecosystem using components such as Services and Deployments. In Logging and Monitoring, a Metric Scraper will periodically obtain relevant metrics and store it into Metric Database. We can further extend with observability tools such as Data Visualiser. The Custom Scaler can obtain the metrics from Logging and Monitoring and feed it into the Metric Gatherer. It will then forward the filtered data to the Logic Evaluator to determine the workflow for adaptation if one is necessary. The Adaptation Executor will follow through the workflow and make changes to the Kubernetes cluster using the Kubernetes API.

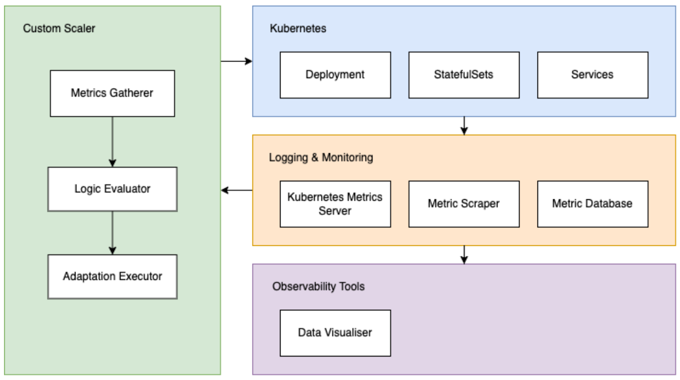

Figure 1: Technical Framework

3.3 Load Testing

A load testing tool such as Locust[16] was utilised to deliberately generate concurrent users that visit the desired HTTP endpoint to simulate user traffic. While

[14]https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar
[15]https://istio.io/
[16]https://locust.io/

load testing the system, Locust will provide numerous of metrics in real-time to show the performance of the system. Requests per second (RPS) can be observed to gauge the overall throughput while the average 90th percentile request duration (p90) can be observed to better understand how long does 90 percent of users wait before the server fulfills their requests.

4 EXPERIMENTAL SETUP

The Kubernetes cluster is using a local machine with Minikube[17]. The current configurations 6 CPU cores with 8192 MB of RAM. The VM driver powering Minikube is Hyperkit and we have additionally configured a setting to prevent Linux open files issue in the underlying Docker runtime engine. More details can be found at the Source Code subsection.

Our experiment will employ to investigate if there is a difference between the average HTTP request duration between HPA and Kubernetes Operators. Let μ_{HPA} be the described metric for HPA and let μ_{KO} be the described metric for Kubernetes Operators. Our null and alternative hypothesis are defined as follows:

$$H_0 : \mu_{HPA} - \mu_{KO} = 0$$
$$H_A : \mu_{HPA} - \mu_{KO} \neq 0$$

4.1 System-wide Adaptation Goals

Adaptation Goals	Adaptation Actions
Ensure the connections per pod is between 20 to 100	Scale pods down if below 20, scale pods up if above 100
Ensure the HTTP error rate is between 0 to 0.3	Scale pods down if below 0, scale pods up if above 0.3
Ensure the average CPU usage per second is between 0.003 to 0.005	Scale pods down if below 0.003, scale pods up if above 0.005
Ensure the average memory usage (MB) is between 340MB and 380MB	Scale pods down if below 340MB, scale pods up if above 380MB

Table 1: Adaptation Goals

In our prototype, we have defined the adaptation goals as according to Table 1. With our custom defined goals, we will scale the pods down or up using an executor that will be defined in later sections.

Technical Component	Description
Nginx Load Balancer	L4 load balancer to distribute traffic to pods
Node.js Application	Web application with REST APIs
MongoDB ReplicaSet	NoSQL database to read and write data
Prometheus	Metrics collection tool by periodically scraping certain endpoints
Grafana	Data visualisation based on Prometheus data
Kubernetes Custom Operator	Custom solution to handle self-adaptation tasks

Table 2: Breakdown Of Technical Components

REST APIs	Description
GET /	Landing Page
GET /auth/login	Login Callback from Auth0
GET /results	Results page
POST /metrics	Metrics page for Prometheus to scrape
POST /requests/create	Submit a request remark for a specific course

Table 3: Application API Routes

Custom Metrics Description	Prometheus Query Language
Number of connections per pod over two minutes	sum(avg_over_time(nginx_ingress_controller_nginx_process_connections{state="active"}[2m]))
HTTP error rate (4xx & 5xx error codes) over two minutes	sum(rate(nginx_ingress_controller_requests{status=~"[4—5].*"}[2m]))/sum(rate(nginx_ingress_controller_requests[2m]))
Average CPU usage per second over two minutes	avg(rate(nginx_ingress_controller_nginx_process_cpu_seconds_total[2m]))
Average Memory usage (MB)	avg(nginx_ingress_controller_nginx_process_resident_memory_bytes)

Table 4: Custom Metrics using PromQL

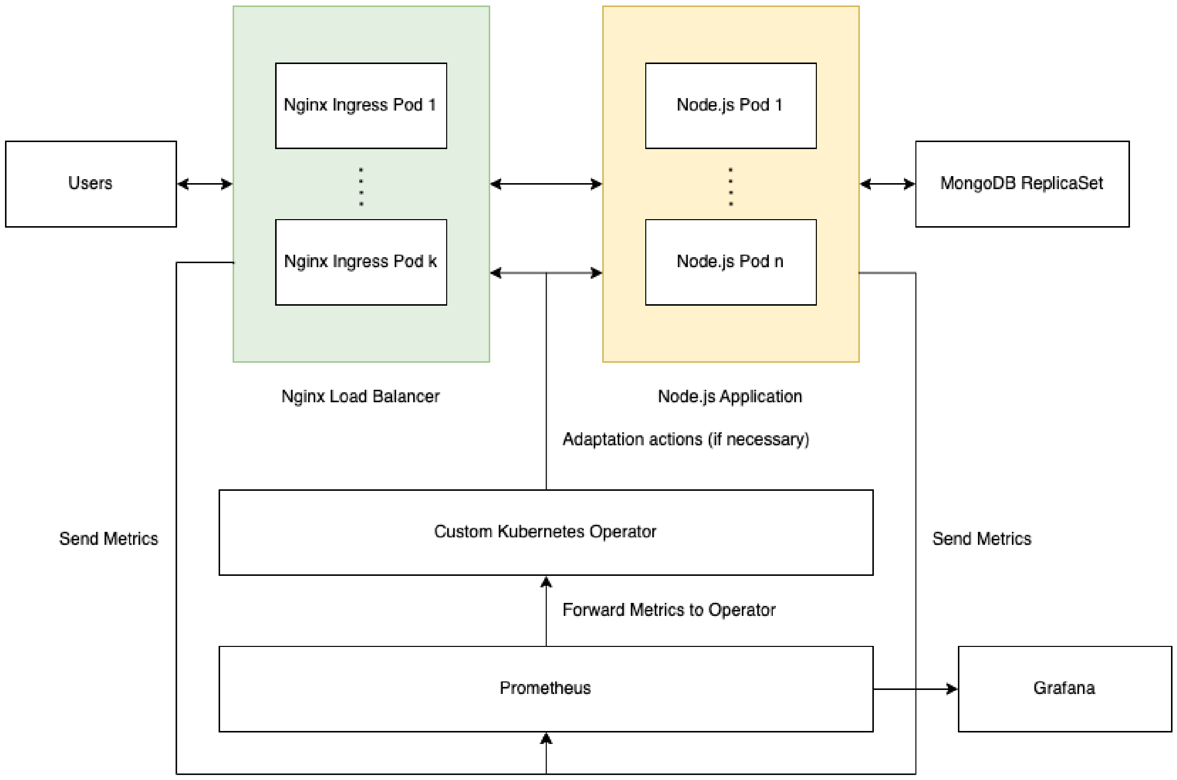

Figure 2: Technical Architecture

4.2 System Architecture

A prototype has been developed based on the reference to Figure 1. The implemented components and their respective descriptions can be found at Table 2. The full list of API routes are listed in Table 3.

By default, we will have one pod for all technical components described except for MongoDB ReplicaSet. In order to have a fallback for the data, we will bootstrap MongoDB ReplicaSet with 2 replicas.

With reference to Figure 2, users can first make HTTP requests. Then, the individual Nginx Ingress Pods in the Nginx Load Balancer will distribute the traffic equally to the relevant Node.js Deployment. The individual Node.js application is then able to run database operations against the MongoDB ReplicaSet. Once the request has been resolved, a HTTP status code and response will be forwarded back to the user.

The Nginx Ingress Pods and Node.js Pods will consistently send metrics to Prometheus. For administrators, they are able to visualise real-time data on Grafana dashboards. The executor involved in self-adaptation is the Kubernetes Custom Operator. The main responsiblity involved is to execute the necessary adaptation tasks to ensure adaptation goals are always met. The Prometheus Query Language (PromQL) used to obtain the custom metrics are defined in Table 4. If a change to the Kubernetes cluster is requested, the Operator will invoke Kubernetes API to scale the respective Pods.

4.3 Custom Autoscaler with Kubernetes Operator

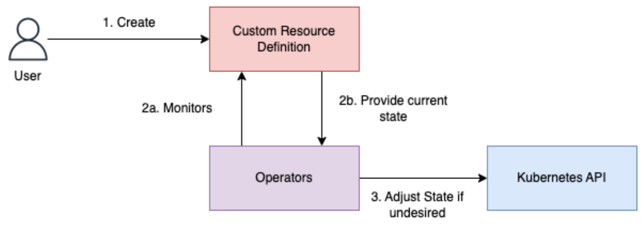
User
1. Create
Custom Resource Definition
2a. Monitors
2b. Provide current state
Operators
3. Adjust State if undesired
Kubernetes API

Figure 3: Kubernetes Operators

[17]https://minikube.sigs.k8s.io/docs/start/

The Kubernetes Operator is written in Go[18]. Kubernetes was written in Go, therefore open-source Go-based libraries are available to communicate with the Kubernetes API to speed up development.

First, a Kubernetes Custom Resource Definition (CRD) must be created. In our example, we created a CRD named CustomScaler. In order to monitor the CRD, a custom operator that synchronises with the Kubernetes reconcile loop is created. As our CustomScaler is able to modify the Nginx Load Balancer Deployment and Node.js Deployment, we can explicitly state that our CRD owns Deployments to prevent collision due to conflict of interest. By invoking the proper APIs using the PromQL defined in Table 4, we are able to programmatically alter the state of the Kubernetes cluster.

4.4 Performance Benchmarking

Benchmarking is primarily performed using a load-testing tool called K6[19] by writing short JavaScript files to define parameters and testing workflows. In all of our load testing, we have used the following parameters:

- Spawn rate of 2000 users per second
- Maximum of 2000 concurrent users per second
- Testing duration of 120 seconds

We have performed 10 iterations based on following three variations. More details can be found in the Results section.

(1) Default Kubernetes (As control)
(2) HPA that scales when CPU Utilisation is 50%
(3) Kubernetes Operators with adaptations goals defined in Table 1

5 EXPERIMENTAL RESULTS

5.1 Results

Based on Table 5, Kubernetes Operators is a clear winner for all metrics except the minimum HTTP request duration in seconds. HPA performs worst than the default Kubernetes for all metrics except maximum HTTP request duration in seconds. By comparing the average, p(90) and p(95) metrics between Kubernetes Operators and HPA, Kubernetes Operators achieved a better performance of 31.04%, 30.02% and 43.87% respectively.

At a significance level of 5%, we can perform a t-test to investigate the statistical significance in our hypothesis, as we are uncertain of the true population variance. First, we officially define μ_{HPA} and μ_{KO} to be the average HTTP request duration in seconds, as listed in Table 5. Next, we calculate the p-value which is <0.001. This means that there is sufficient evidence to reject the null hypothesis at 5% significance level and this suggest that the mean average HTTP duration in seconds between HPA and Kubernetes Operators are not the same.

[18]https://go.dev/
[19]https://k6.io/

5.2 Discussion

One possibility of why the HPA performed the worst amongst all the variation is the usage of one runtime metric which is CPU. With only one runtime metric, the scope to perform self-adaptation is highly limited and there could be other metrics that can better represent the actual performance of the system. However, due to the limited support for other metrics, HPA have the worst performance.

The introduction of a component to perform self-adaptations could introduce additional latency onto the system. Due to the need of pods organisation and resource watching, the system will experience more data transmissions for monitoring. Hence, this might be a possibility on why the HPA performs worse than the control variation.

5.3 Visualisation of Kubernetes Operators During Runtime

Locust has been chosen as it offers a realtime visualisation during the load testing. The parameters used in this test are the same as the previous defined parameters. Based on Figure 4, there is two huge spike for p(95). However, the p(95) reduces after that and stabilises with the median response time. This is because the Kubernetes Operator triggered a series of self-adaptation actions in the background while it follows our predefined adaptation goals.

5.4 Source Code

All source code can be found at a Github repository found at https://github.cs.adelaide.edu.au/a1834032/topics. A Readme file has also been attached to provide installation instructions. The full data obtained based on the 10 iterations can be found in a .csv file in the repository.

6 LIMITATIONS

In our prototype, we have demonstrated a use case of which the Custom Kubernetes Operator is successful

	HTTP Request Duration in seconds						Requests	
	Average	Minimum	Median	Maximum	p(90)	p(95)	Total Requests	Requests Per Second
Control	6.81	**0.33**	6.57	29.59	9.82	12.40	31118.50	257.90
HPA	7.43	0.61	7.25	27.63	10.09	14.79	29247.80	246.21
Kubernetes Operators	**6.30**	0.47	**6.25**	**20.31**	**8.47**	**11.60**	**36569.60**	**301.06**

Table 5: Average data collected based on 10 iterations.

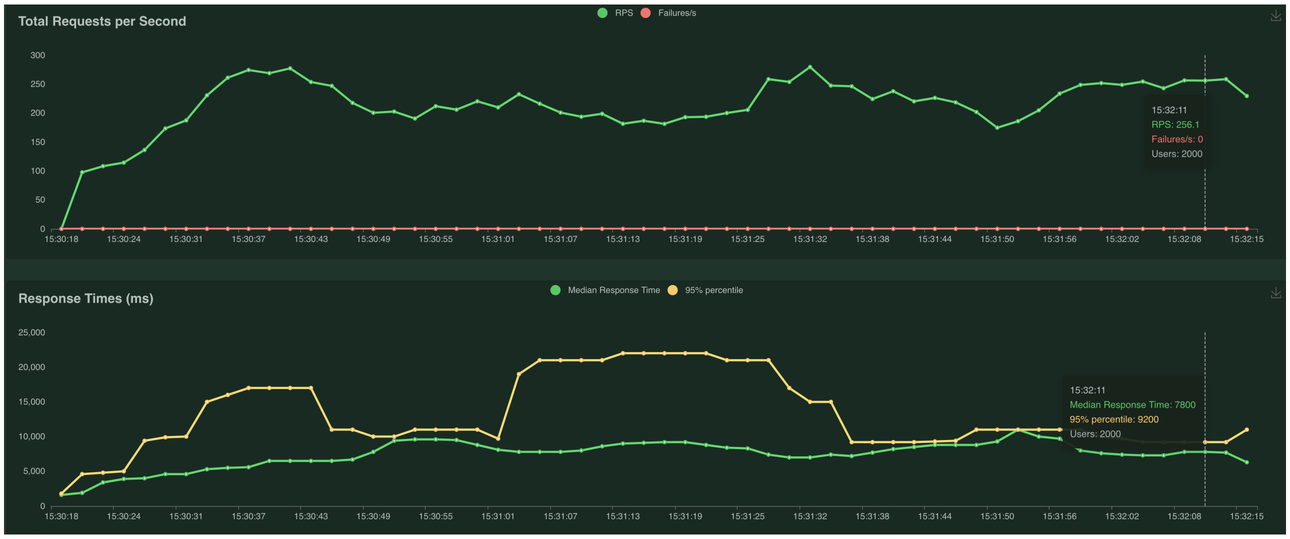

Figure 4: Locust real-time graph using Kubernetes Operators.

in performing self-adaptations based on custom adaptation goals. However, there are certain limitations that will be discussed.

6.1 MongoDB Scaling

Our prototype did not implement any scaling for MongoDB. Based on Table 5, the HTTP request duration for average, p(90) and p(95) is high. Due to the high read and write operations during load testing, there is a possibility that the bottleneck could be the MongoDB ReplicaSet. The current ReplicaSet approach utilises a primary node that is responsible for all write operations and most read operations. A possible solution to this problem is to introduce a database cache layer such as Redis[20] to reduce the read operations needed for each client request. Another possible solution is to use MongoDB Sharding and horizontally scale it.

6.2 Prometheus Scrape Intervals

The introduction of metric collection will incur some CPU resources and add latency to our system. With the constant need to scrape metrics, the amount of API calls will increase constantly according to the number of microservices that we expose to Prometheus. There is a tradeoff between scrape interval, the time of which the self-adaptation can take place and the additional latency. By default, Prometheus is configured to scrape metrics every 30 seconds. A shorter scrape interval will lead to a faster self-adaptation, but this will incur more resources and increase latency. The opposite effect applies.

7 FUTURE WORK

Threshold-based adaptation goals require tuning of parameters to allow HPA and Kubernetes Operators to operate optimally. However, the manual tuning to obtain a suitable threshold is not trivial. Future work will explore ways to potentially hasten the process while benchmarking against other methods that have been researched such as reinforcement learning.

8 CONCLUSION

Kubernetes Operators allows us to extend Kubernetes to perform self-adaptations with custom metrics. First,

[20]https://redis.io/

we define several adaptation goals and demonstrate our prototype. With the usage of different open-source tools such as Kubernetes, the prototype has shown to be realistic and practical for current and future projects. Then, a statistical significance has shown that there is a difference in the mean average HTTP duration in seconds between HPA and Kubernetes Operators. The prototype developed and experiment carried out have demonstrated that custom metrics have proven to increase runtime performance and developers can easily integrate custom metrics with Kubernetes Operators.

REFERENCES

[1] Carlos M Aderaldo, Nabor C Mendonça, Bradley Schmerl, and David Garlan. 2019. Kubow: An architecture-based self-adaptation service for cloud native applications. In *Proceedings of the 13th European Conference on Software Architecture-Volume 2*. 42–45.

[2] Shang-Wen Cheng and David Garlan. 2022. Handling uncertainty in autonomic systems. (10 2022).

[3] Martina De Sanctis, Henry Muccini, and Karthik Vaidhyanathan. 2020. Data-driven adaptation in microservice-based iot architectures. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 59–62.

[4] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 37, 10 (2004), 46–54.

[5] Omid Gheibi, Danny Weyns, and Federico Quin. 2021. Applying Machine Learning in Self-Adaptive Systems: A Systematic Literature Review. *ACM Trans. Auton. Adapt. Syst.* 15, 3, Article 9 (aug 2021), 37 pages. https://doi.org/10.1145/3469440

[6] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-Driven Autoscaling for Microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1994–2004. https://doi.org/10.1109/ICDCS.2019.00197

[7] Shay Horovitz and Yair Arian. 2018. Efficient Cloud Auto-Scaling with SLA Objective Using Q-Learning. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*. 85–92. https://doi.org/10.1109/FiCloud.2018.00020

[8] DataDog LLC. 2021. 10 Trends in Real-world Container Use. (2021). https://www.datadoghq.com/container-report/

[9] Nabor Mendonca and Carlos Aderaldo. 2021. Towards First-Class Architectural Connectors: The Case for Self-Adaptive Service Meshes. In *Brazilian Symposium on Software Engineering*. 404–409.

[10] Nabor C Mendonça, David Garlan, Bradley Schmerl, and Javier Cámara. 2018. Generality vs. reusability in architecture-based self-adaptation: The case for self-adaptive microservices. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. 1–6.

[11] William Morgan. 2019. The Service Mesh: What Every Software Engineer Needs to Know about the World's Most Over-Hyped Technology. *URL: https://buoyant.io/servicemesh-manifesto (visited on 2021-05-07)* (2019).

[12] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. 1999. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications* 14, 3 (1999), 54–62.

[13] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. 2019. Cloud Container Technologies: A State-of-the-Art Review. *IEEE Transactions on Cloud Computing* 7, 3 (2019), 677–692. https://doi.org/10.1109/TCC.2017.2702586

[14] Fabiana Rossi. 2020. Auto-scaling Policies to Adapt the Application Deployment in Kubernetes.

[15] Areeg Samir and Claus Pahl. 2019. A Controller Architecture for Anomaly Detection, Root Cause Analysis and Self-Adaptation for Cluster Architectures.

[16] Upendra Sharma, Prashant Shenoy, and Donald Towsley. 2012. Provisioning multi-tier cloud applications using statistical bounds on sojourn time. *ICAC'12 - Proceedings of the 9th ACM International Conference on Autonomic Computing*, 43–52. https://doi.org/10.1145/2371536.2371545

[17] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html