Kubow: An Architecture-Based Self-Adaptation Service for Cloud Native Applications

Carlos M. Aderaldo
Nabor C. Mendonça
carlosmendes@unifor.br
nabor@unifor.br
University of Fortaleza
Fortaleza, CE, Brazil

Bradley Schmerl
David Garlan
schmerl@cs.cmu.edu
garlan@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

ABSTRACT

This paper presents Kubow, an extensible architecture-based self-adaptation service for cloud native applications. Kubow itself was implemented by customizing and extending the Rainbow self-adaptation framework with support for Docker containers and Kubernetes. The paper highlights Kubow's architecture and main design decisions, and illustrates its use and configuration through a simple example. An accompanying demo video is available at the project's web site: https://ppgia-unifor.github.io/kubow/.

CCS CONCEPTS

• **Software and its engineering → Software infrastructure**.

KEYWORDS

self-adaptation, rainbow, kubernetes

ACM Reference Format:
Carlos M. Aderaldo, Nabor C. Mendonça, Bradley Schmerl, and David Garlan. 2019. Kubow: An Architecture-Based Self-Adaptation Service for Cloud Native Applications. In *13th European Conference on Software Architecture (ECSA '19), September 9–13, 2019, Paris, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The increasing use of lightweight containers, e.g. Docker,[1] and container orchestration tools, e.g., Kubernetes,[2] represents a significant paradigm shift for the adoption of cloud computing technologies, which moves the focus from managing virtual machines to managing containers and services [10]. Kubernetes, in particular, is rapidly becoming the *de facto* standard for managing cloud native applications, and is already being offered as a service by all major cloud providers in the market [1].

A crucial aspect of managing a cloud native application is *self-adaptation*. A system is self-adaptive if it can reflect on its behavior at runtime and change itself in response to environmental conditions, errors, and opportunities for improvements [13]. Typically, self-adaptation capabilities are provided to some target system by adding a self-adaptation layer that reasons about observations of the system's runtime behavior, decides whether the system is operating outside its required bounds and what changes should be made to restore the system, and effects those changes on the system. This form of self-adaptation essentially involves adding a closed control loop layer onto the system. A typical self-adaptation control loop consists of four main activities: Monitor, Analyze, Plan, and Execute, all sharing a common Knowledge base, usually referred to as the MAPE-K reference model [8].

Although there is a vast literature on methods, techniques and tools for self-adaptation, only a handful of self-adaptive solutions, such as automated server management and cloud elasticity, have thus far found their way to industrial applications [13]. One possible explanation is that most of the existing self-adaptation solutions have been proposed targeting traditional (i.e., on-premise) production environments. As a consequence, traditional architecture-based self-adaptation frameworks, such as Rainbow [5], which have been developed with reusability and extensibility as key design principles, in practice have proved to be notoriously difficult to customize for modern production environments that rely heavily on container-based deployment and management technologies [9].

In an attempt to fill this gap, this paper presents Kubow, an extensible architecture-based self-adaptation service for cloud native applications. Kubow itself was implemented by customizing and extending Rainbow with support for Docker containers and Kubernetes. Thus far Kubow has successfully been used to manage existing and new Kubernetes applications in both local and cloud-based environments.

The remainder of the paper describes Kubow's self-adaptive architecture and main design decisions (Section 2); illustrates its use and configuration through an example (Section 3); compares it with related solutions (Section 4); and states our conclusions (Section 5).

2 KUBOW OVERVIEW

Kubow provides a suite of technologies that allow augmenting an existing Kubernetes application with MAPE capabilities. To this end, Kubow customizes and extends Rainbow's original MAPE-like components [5]. Those components include (see Figure 1): a *Model Manager*, which stores the target application's architecture model and mediates the interactions between the other components;

[1] https://www.docker.com/
[2] https://kubernetes.io/

ECSA'19, September 9–13, 2019, Paris, France

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

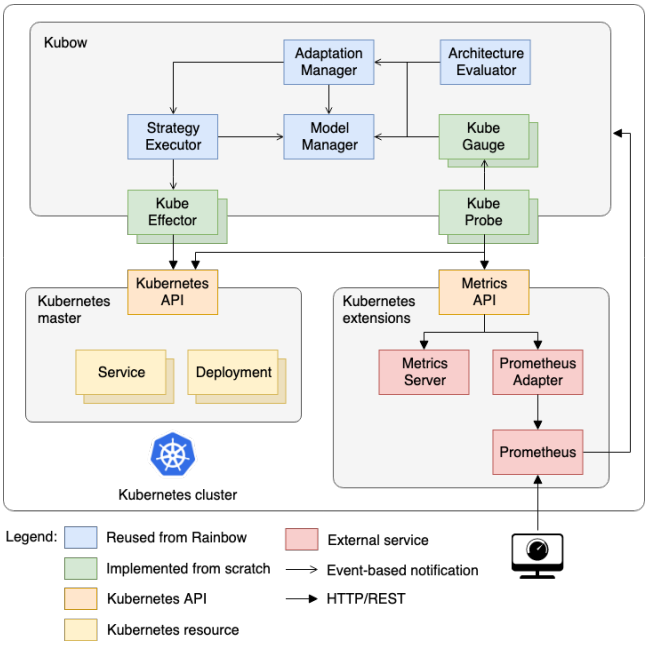

Figure 1: Kubow architecture.

a set of *Probes* and *Gauges*, which monitor the behavior of the target application and update its architecture model's attributes; an *Architecture Evaluator*, which analyzes the target application's state expressed in its architecture model and decides whether any adaptation is necessary; an *Adaptation Manager*, which selects the best adaptation strategy from a set of user-provided strategies, tactics, and utility functions; a *Strategy Executor*, which manages the execution of the selected adaptation strategy; and a set of *Effectors*, which implement the mechanisms necessary to change the target application's behavior in its execution environment.

At execution time, Rainbow components are deployed into two separate units, called *Master* (composed of the Model Manager, the Architecture Evaluator, the Adaptation Manager, and the Strategy Executor) and *Delegate* (composed of Probes, Gauges and Effectors). The Master components are application-independent and deployed in a centralized fashion. The Delegate components are application-specific and can have multiple instances, which are deployed in and share the same execution environment of the target application. All Rainbow components both within and across those two deployment units communicate asynchronously via an event-driven message oriented middleware.

In addition to the original Rainbow components, Kubow uses several specific Kubernetes services to implement its Probes and Effectors. Those services are invoked by means of two APIs: *Kubernetes API*, which is the main way to access the different types of resources managed by Kubernetes (e.g., Pods, Deployments, Services); and *Metrics API*, which extends the Kubernetes API to provide a uniform way to access metrics collected by Kubernetes' own monitoring services (e.g., Kubelet, cAdvisor) as well as by other external monitoring tools (e.g., Prometheus[3]).

[3]https://prometheus.io/

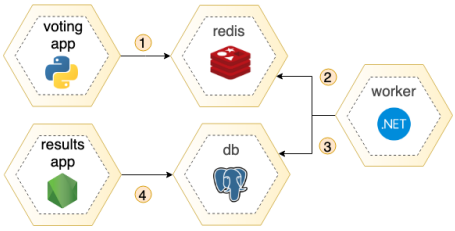

Figure 2: Voting app architecture.

The need to integrate with Kubernetes resulted in two major design decisions that set Kubow apart from Rainbow's original architectural design. The first decision was to implement all Probes and Effectors in an application-independent way. The second decision was to package both Master and Delegate components into a single container image, which is then deployed and managed as a regular Kubernetes service. Both decisions were made possible by the fact that Kubernetes provides common monitoring and management APIs. Thus, by simply changing the Kubernetes invocation parameters, the same set of Probes and Effectors can be reused for monitoring and managing, respectively, different application components. This also means that there is no longer the need to deploy separate Delegates alongside the target application nodes, as all application resources in Kubernetes can be managed remotely by invoking the appropriate Kubernetes APIs.

3 EXAMPLE OF USE

To illustrate Kubow's use, we selected a simple web-based voting app from the set of sample applications distributed with Docker.[4] This application was chosen because its architecture is simple enough to be used for illustrative purposes, and complex enough to be representative of typical Kubernetes applications deployed in real production environments.

The voting app architecture comprises five components, each one implemented using a different programming languages or storage technology, as shown in Figure 2. These five components work as follows: (1) a web-based voting app (voting-app), implemented in Python, collects the user votes and publishes them to a message queue (redis), implemented in Redis; (2) a background voting processor (worker), implemented in .NET, counts the votes being published to the message queue and (3) stores the (partial) voting results in a Postgres relational database (db); and, finally, (4) a second web-based app (result-app), implemented in Node.js, reads the voting results from the relational database and shows them to the voting application users in real time.

3.1 Architecture Model

The first step to integrate Kubow with an existing Kubernetes application is to define the application's architecture model in terms of its sets of containerized services and connectors. As with Rainbow, Kubow architecture models are defined in Acme, a well-known architecture description language [6].

Figure 3 shows the Acme specification for the result-app and db components. That specification refers to two component types,

[4]https://github.com/dockersamples/example-voting-app

```
1 import families/Kubernetes.acme
2 System VotingAppSystem: KubernetesFam = new
      KubernetesFam extended with {
3   Component resultD = new DeploymentT extended with {
4     Property name = "result";
5     Property namespace = "votingapp";
6   }
7   Component resultS = new ServiceT extended with {
8     Property name = "result";
9   }
10  Component dbD = new DeploymentT extended with {
11    Property name = "db";
12    Property namespace = "votingapp";
13  }
14  Component dbS = new ServiceT extended with {
15    Property name = "db";
16  }
17  Connector resultSC = new LabelSelectorConnectorT
        extended with {
18    Property selectors = <[name: string = "app"; value
          : string = "result";]>;
19  }
20  Connector dbSC = new LabelSelectorConnectorT
        extended with {
21    Property selectors = <[name: string = "app"; value
          : string = "db";]>;
22  }
23  Connector resultDbConn  = new ServiceConnectorT
        extended with {
24    Property namespace = "votingapp";
25    Property name = "result";
26  }
27  Attachment resultS.redirectPort to resultSC.callee;
28  Attachment resultD.redirectPort to resultSC.caller;
29  Attachment dbS.redirectPort to dbSC.callee;
30  Attachment dbD.redirectPort to dbSC.caller;
31  Attachment resultD.sqlPort to resultDbConn.selector;
32  Attachment dbS.sqlPort to resultDbConn.selectee;
33 }
```

Figure 3: Acme specification for result-app and db.

namely DeploymentT (lines 3 and 10) and ServiceT (lines 7 and 14), and two connector types, namely LabelSelectorConnectorT (lines 17 and 20) and ServiceConnectorT (line 23), which are all part of the Acme family previously defined in Kubow for a representative subset of the Kubernetes application domain.[5] Note that the family specification must be explicitly imported in the beginning of the architecture model specification (line 1).

Those two component types represent two typical Kubernetes resource types, namely, Services and Deployments. The two connector types, in turn, represent two rather distinct ways of establishing explicit associations between different Service and Deployment resources in Kubernetes. The LabelSelectorConnectorT connector type represents the definition of a label selector in Kubernetes, which, in this particular scenario, is used to establish a load balancing association between the result-app and db Services and their corresponding Deployments (lines 27-28 and 29-30, respectively). The ServiceConnectorT connector type, on the other hand, represents a typical invocation association between a Deployment and a Service. In this scenario, a single ServiceConnectorT connector is used to establish an invocation association between the result-app Deployment and the database Service (lines 31-32). The other three components were specified in a similar manner.

[5]Kubernetes also supports other types of applications, such as cron jobs, which are typically used to run time-based batch processing tasks. We intend to extend the Acme family model of Kubow to support these other application types in our future work.

```
1 define boolean cHiRespTime = M.resultS.latency > M.
      MAX_RESPTIME;
2 define boolean cLoRespTime = M.resultS.latency < M.
      MIN_RESPTIME;
3 define boolean canAddPods = M.resultD.maxReplics > M.
      resultD.desiredReplics;
4 define boolean canRemovePods = M.resultD.minReplics <
      M.resultD.desiredReplics;
5 tactic addReplicas(int count) {
6   condition {
7     cHiRespTime && canAddPods;
8   }
9   action {
10    M.scaleUp(M.resultD, M.resultD.desiredReplics +
          count);
11  }
12  effect {
13    M.resultD.maxReplics >= M.resultD.desiredReplics;
14  }
15 }
16 tactic removeReplicas(int count) {
17  condition {
18    cLoRespTime && canRemovePods;
19  }
20  action {
21    M.scaleDown(M.resultD, M.resultD.desiredReplics -
          count);
22  }
23  effect {
24    M.resultD.minReplics <= M.resultD.desiredReplics;
25  }
26 }
```

Figure 4: Stitch specification for the result-app tactics.

From the definition of the target application's architecture model follows the definition of the other Kubow elements necessary for monitoring (Probes and Gauges), analyzing (tactics, strategies and utility functions) and changing (Effectors) the application's behavior at execution time. Due to space limitations, here we only show snippets of the tactics and strategies defined for the result-app component. More details on the design, implementation, use and evaluation of Kubow will be provided in a future publication.

3.2 Tactics and Strategies

Again, as in Rainbow, in Kubow tactics and strategies are defined using Stitch, a domain specific language for self-adaptation frameworks [2]. Figure 4 shows the Stitch specification for the two tactics defined for result-app, namely addReplicas() (lines 5-15) and removeReplicas() (lines 16-26), which attempt to add and remove Pods to/from a Kubernetes Deployment, respectively. The "M" in the code shown in Figure 4 as well as in Figure 5 refers to the application's architecture model.

Note that each of those tactics invokes a separate adaption operation, namely scaleUP() (line 10) and scaleDown() (line 21), directly on the model element representing the service to be adapted. At

```
1 strategy ReduceRespTime [ cHiRespTime ] {
2   t0: (cHiRespTime && canAddPods) -> addReplicas(1) @
        [30000 /*ms*/] {
3     t0a: (success) -> done;
4   }
5   t2: (default) -> TNULL;
6 }
7 strategy ReduceCost [ cLoRespTime ] {
8   t0: (cLoRespTime && canRemovePods) -> removeReplicas
        (1) @[30000 /*ms*/] {
9     t0a: (success) -> done;
10  }
11  t1: (default) -> TNULL;
12 }
```

Figure 5: Stitch specification for the result-app strategies.

execution time, those operations are handled by the Strategy Executor, which invokes the specific Kubow Effectors responsible for creation and removal of Pods, respectively.

Figure 5, in turn, shows the Stitch specification for the two strategies defined for result-app, namely ReduceRespTime (lines 1-6) and ReduceCost (lines 7-12), which attempt to reduce the service's response time and cost, respectively. Since these two strategies refer to the same Stitch variables defined in the tactics specification (Figure 4), the declaration of those variables has been omitted from Figure 5.

Note how the ReduceRespTime strategy invokes the addReplicas() tactic only when the service response time is above the maximum threshold, and it is still possible to add new Pods (line 2). Similarly, the ReduceCost strategy invokes the removeReplicas() tactic only when the service response time is below the minimum threshold, and it is still possible to remove existing Pods (line 8). Note also that both strategies define a 3 second invocation latency for each tactic, which corresponds to the time interval the Strategy Executor will have to wait before checking whether the tactic execution was successful.

The demo video accompanying this paper shows how Kubow can be used to monitor and preserve the response time of an existing Kubernetes application under varying load conditions.

4 RELATED WORK

There is an emerging body of work proposing self-adaptation solutions for containerized applications. In [4], the authors describe a decentralized approach for horizontal auto scaling of microservice applications [7], which is implemented as a new abstraction layer on top of Docker. In [3], the authors present an approach for implementing and evaluating different autoscaling solutions for replicated database clusters deployed in Kubernetes. That approach is implemented by extending Kubernetes's native replication controllers. In [12], the authors propose an approach for dynamic reallocation of containerized microservices, which is implemented by extending Kubernetes's native container scheduler. Finally, in [11], the authors describe a novel container management mechanism for Kubernetes, whose goal is to dynamically consolidate application containers in a minimal number of nodes, so as to avoid unnecessary scaling of the underlying virtual infrastructure.

The solutions described above have a limited adaptation scope in the sense that they only support a fixed set of adaptation mechanisms (e.g., container reallocation and auto scaling). The same criticism applies to most (if not all) self-adaption tools currently used by industry. These include cloud native auto scaling services, such as Amazon's CloudWatch,[6] and Kubernetes's own Horizontal Pod Autoscale (HPA) controller.[7] Although some of those solutions do support the creation and customization of application-specific monitoring metrics, their native set of adaptation mechanisms cannot be extended by application developers.

In contrast to all the above solutions, Kubow builds on Rainbow to provide an extensible self-adaptation solution for Kubernetes applications. Even though this paper only shows examples of tactics and strategies dealing with auto scaling, Kubow is a generic solution that allows the easy creation and reuse of a variety of adaption mechanisms, tactics and strategies, in a declarative way, without the need to change its source code.

5 CONCLUSION

This paper presented Kubow, an extensible self-adaptation service for containerized Kubernetes applications. Kubow is freely available at https://github.com/ppgia-unifor/kubow/. We hope its public release can contribute to foster a more systematic development and reuse of self-adaptation solutions by the software architecture and cloud computing communities.

ACKNOWLEDGMENTS

This work is partially supported by INES (www.ines.org.br), CNPq grants 465614/2014-0, 313553/2017-3 and 424160/2018-8, FACEPE grant APQ-0399-1.03/17, and PRONEX grant APQ/0388-1.03/14.

REFERENCES

[1] Eric Brewer. 2018. Kubernetes and the New Cloud. In *SIGMOD 2018.* Invited Keynote.
[2] Shang-Wen Cheng and David Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software* 85, 12 (2012), 2860–2875.
[3] Wito Delnat et al. 2018. K8-Scalar: a workbench to compare autoscalers for container-orchestrated database clusters. In *IEEE/ACM SEAMS 2018.* IEEE, 33–39.
[4] Luca Florio and Elisabetta Di Nitto. 2016. Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices architectures. In *IEEE ICAC 2016.* 357–362.
[5] D. Garlan et al. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer* 37, 10 (2004), 46–54.
[6] David Garlan, Robert T Monroe, and David Wile. 2000. Acme: Architectural description of component-based systems. *Foundations of component-based systems* 68 (2000), 47–68.
[7] P. Jamshidi et al. 2018. Microservices: The Journey So Far and Challenges Ahead. *IEEE Software* 35, 3 (2018), 24–35.
[8] Jeffrey O Kephart and David M Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (2003), 41–50.
[9] Nabor C Mendonça et al. 2018. Generality vs. Reusability in Architecture-Based Self-Adaptation: The Case for Self-Adaptive Microservices. In *AKSAS 2018.*
[10] Claus Pahl et al. 2017. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing* (2017).
[11] Maria A Rodriguez and Rajkumar Buyya. 2018. Containers Orchestration with Cost-Efficient Autoscaling in Cloud Computing Environments. *arXiv preprint arXiv:1812.00300* (2018).
[12] Adalberto Ribeiro Sampaio Jr et al. 2018. Improving Microservice-based Applications with Runtime Placement Adaptation. *Journal of Internet Services and Applications* (2018).
[13] D. Weyns. 2017. *Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges.* Springer.

[6] https://aws.amazon.com/cloudwatch/
[7] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/