Kubernetes Operators as a control system for cloud-native applications

Fedor Y. Chemashkin[#1], Pavel D. Drobintsev [#2]
[#]The High School of Software Engineering, Institute of Computer Science and Technology
Peter the Great St. Petersburg Polytechnic University
St. Petersburg, Russia
[1]fedor.chemashkin@gmail.com
[2]drob@ics2.ecd.spbstu.ru

***Abstract*— Cloud-native architecture was developed to deal with rapidly changing and uncertain cloud environments. Cloud-native applications have strict managing requirements and by implementing cloud-native architecture it becomes easier to develop and control such software. Cloud-native applications are usually managed by Kubernetes. It allows using the controller pattern that came from the control theory for control software. With Kubernetes' possibility to extend a developer can create own controller with specific knowledge about their software. Control theory can help to make a controller for an application that will be mathematically proved by control theory methods and laws.**

Keywords— cloud-native, cloud-native architecture, cloud-native applications, kubernetes; operators, controllers, devops, control systems, self-adaptive systems.

I. INTRODUCTION

Increasingly, software engineers are choosing a cloud environment as a platform to launch their software systems. Cloud environments provide a lot of useful functionality that can increase economical venue and business value even for a small application. However, with many advantages, clouds have their own cost that must be considered at the stage of making the decision to run an application in a cloud.

Nowadays many companies consider the cost and many successful experiences of other companies and decide to move to the clouds.

For leveraging all the advantages and functionally of cloud environments, there are many paths to do it. In [1] it is called "7R's of migration".

"7R's of migration" contains:

- replacing an existing application with a SaaS service or creating a new one with cloud-native characteristics and features;
- reusing common business and technical services offered by other vendors;
- refactoring an existing application according to a cloud-native architecture.
- re-platforming an application environment and integration with cloud features;
- rehosting an application with minimal changes in its source code (if it possible);
- retaining an application as is;
- retiring an application lifecycle, i.e. end of life.

Based on our experience we can conclude that refactoring and replacing are the hardest but the most promising way to become cloud-native that can give a lot of benefits in the future.

The industry and academia have already defined cloud-native architecture that can guide software engineers while creating cloud-native applications.

Cloud-native applications are executed in unpredictable environments and applications' requirements can be changed in runtime depending on customers' needs. Control Theory can be used to address these issues.

This idea was explored in [2], [3], and [4]. These papers contain possible ways to apply Control Theory in software engineering. However, they focused on some abstract examples of software systems and perhaps this leads to the fact that it becomes not entirely clear how to apply it in practice.

In our work, we have started to research cloud-native architecture and applications and their adaptive properties. The industry concluded that it is difficult to run complex stateful applications in cloud environments and therefore introduced a new pattern for deploying such applications in Kubernetes – Kubernetes operators. Kubernetes operators are control systems for complex stateful applications (Section 4-6).

This work is a Research-in-Progress paper and contains interim results of research of cloud-native architectures and applications and possible application of control theory for such use cases. In this paper, our main goals are to describe cloud-native architecture and application and create a mapping between regular control theory terms and Kubernetes operators concepts.

II. Cloud-native architecture

An architectural style improves the separation of concerns and promotes design reuse by providing solutions to frequently recurring problems [5]. Sometimes an architectural style can be called an architectural pattern.

An architectural pattern includes sets of principles and patterns based on which applications can be developed. The principles describe an architectural design of the entire software system that following this architecture, and the patterns implement the principles and are considered frequently used best practices that confirmed by real operational experience. Also, the definition of an architectural style in some areas of the industry gives non-technical benefit – create a tool for discussions some aspects of software without locking to technologies.

In [6], [7], and [8] cloud-native architecture is defined by using control-theoretic and model-based approaches. This architecture has the following principles:

- Virtualization
- Service-orientation
- Uncertainty
- Adaptivity

Patterns implement these principles to templates that can be used in multiple applications:

- Microservices
- Models at runtime
- Controller-based feedback loop

These principles and patterns allow creating a self-adaptive software system. Self-adaptive systems are necessary to use to carry out changing requirements in unstable environments like a cloud and maintain the required performance.

These requirements should have representation in the system's runtime via dynamic models. It allows the system to be changed during its lifecycle.

Also, applying microservices patterns developers must be able to create continuous integration and continuous delivery pipelines for their applications since microservices are deploying in their own processes and communicating with each other via a communication network.

III. Cloud-native applications

The Cloud Native Computing Foundation (CNCF) was created to help evaluate and make production-ready cloud-native technologies in open source and vendor-neutral ecosystems.

Based on the CNCF Technical Oversight Committee's definition [9] of Cloud Native we can identify key properties of cloud-native applications (CNA) – containerization, microservice architecture, and usage of immutable infrastructure that means that an application and its infrastructure are dynamically managed. Therefore, we can define that CNA as a containerized, microservice-based, and dynamically managed application. Such applications can be deployed in public, private, and hybrid cloud environments.

Also, there are a lot of similar definitions of cloud-native applications. E.g. "A cloud-native application is a distributed, elastic and horizontal scalable system composed of (micro)services which isolate state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform." [10].

A lot of papers and industry experts have an opinion that cloud-native applications require using DevOps practices to take full advantage and reduce the disadvantages of cloud environments. Besides it, using microservices require it.

DevOps practices like continuous integration and deployment, automated acceptance testing, monitoring, logging, and so on allow CNA to be developed much faster, and paradoxically, high-quality.

According to "The Accelerate State Of DevOps Report" the high-performance teams that use DevOps practices develop more qualitative software than teams that don't use these practices.

IV. Kubernetes

Cloud-native applications are containerized and dynamically managed. To achieve dynamic management, software called container orchestrators is used. Every container orchestrator (CO) controls multiple hosts like one entity for the end-user.

Sysdig company made the research called "Sysdig 2019 Container Usage Report: New Kubernetes and security insights" [11]. This research says that 77% of Sysdig's customers use Kubernetes. Also, 7% use Rancher and Openshift that built based on Kubernetes. It can be assumed that this report and similar reports of other companies show the current state of the industry.

So, we can conclude that Kubernetes is the most popular container orchestration system and de-facto the industry standard.

Kubernetes is open-source software and has functionality for automatic deployment, scaling, and control of containerized applications [12]. It also offers functionality for managing computing and network resources for various types of workloads.

Kubernetes is a big and complex software system that has 12 components [13]. However, Kubernetes' essence can be represented in Figure 1. The figure shows how Kubernetes can be represented in its main entities.

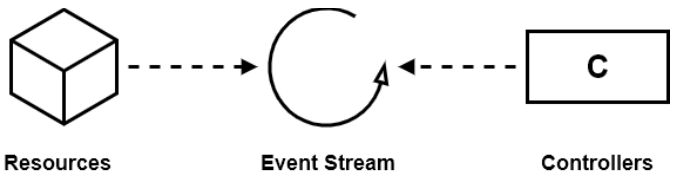

Fig. 1. Kubernetes working mode

In Figure 1, resources are Kubernetes' "building" blocks. Resources have their controllers which implement a control loop that watches for this resource through the Kubernetes API server and performs control actions to move the current state of resources to the desired state.

Sometimes control loop is called the "reconciliation loop" in the Kubernetes ecosystem. Controllers are separate binaries usually written using Golang.

Everything that happens with resources is writing to Kubernetes Event Stream which can be accessed through the API server. The event stream is an append-only log.

It may be said that Kubernetes is not used only for creating cloud-native applications and implementing cloud-native architecture, but also uses this architecture.

V. KUBERNETES EXTENDIBILITY

The Kubernetes' developers laid the possibilities in it to be highly configurable and extensible for minimizing changes into core source code. Configurations are turning on with changing configuration flags, configuration files, and API resources.

Kubernetes has 7 extensions points [14] – original command-line interface by dint of using plugins, API client via new types, API server with custom resources, Kubernetes' behavior with custom controllers for custom resources, Kubernetes scheduler with new rules for scheduling, node-level components via network and storage plugins.

Typically, extensions are new software components that deeply integrate with Kubernetes. It is done to support new resource types and run complex applications.

In this paper, we focus more on custom resources and custom controllers.

In the Kubernetes term, a resource is an endpoint in its default API. It has various built-in resources. A custom resource is an endpoint that extends default endpoints. Besides built-in resources, Kubernetes core functionality is made using custom resources making Kubernetes even more extendable.

After applying a custom resource to the cluster, users can interact with them through CLI as a usual resource. Custom resources can be used as key-value storage since Kubernetes stores information about resources in its etcd instance.

After connecting a custom resource with a custom controller, a custom resource can provide a declarative API that allows declaring the desired state of a resource which can be supported by a custom controller.

These functionalities were the beginning of the creation of a new type of software called "Kubernetes operators".

VI. KUBERNETES OPERATORS

Kubernetes Operator is a combination of custom resources and custom controllers that encode domain-specific knowledge for application as an extension of the Kubernetes API. The emergence of a new role called "Site Reliability Engineer" has led to the fact that applications have been supplemented by domain operational knowledge for better operation in production.

The first time, operators were represented in the CoreOS article "Introducing Operators: Putting Operational Knowledge into Software". In this article, there is the authors' definition of this software – "an Operator is an application-specific controller that extends the Kubernetes API to create, configure, and manage instances of complex stateful applications on behalf of a Kubernetes user. It builds upon the basic Kubernetes resource and controller concepts but includes domain or application-specific knowledge to automate common tasks" [15].

Creating Kubernetes operators was the answer to the challenge of managing large stateful applications, e.g. databases and monitoring systems. Such software requires specific knowledge for correct scaling decisions, upgrade procedures, and numerous configurations to maintain the required application state and performance.

The operator pattern is an implementation of the controller pattern in Kubernetes with some application-specific knowledge. The controller pattern has much greater adaptations both in cloud-native architecture and in simple software without any specifics.

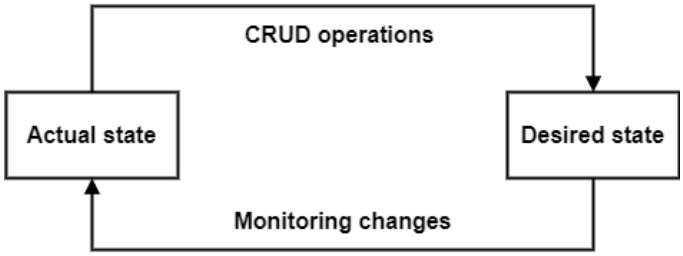

Fig. 2. Controller pattern in Kubernetes

Figure 2 shows how the controller pattern works – read the actual state of a resource from the API server, change the state using CRUD operations and update information about the state in the API server. This reconciliation loop is infinite by its nature.

VII. AUTOMATIC CONTROL OF SOFTWARE AND ADAPTIVENESS

The controller pattern came from the control theory and robotics where a control loop is also a non-terminating loop that regulates (control) the system work. "In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the API server and makes changes attempting to move the current state towards the desired state" [16].

Figure 3 shows a simplified high-level diagram of such systems.

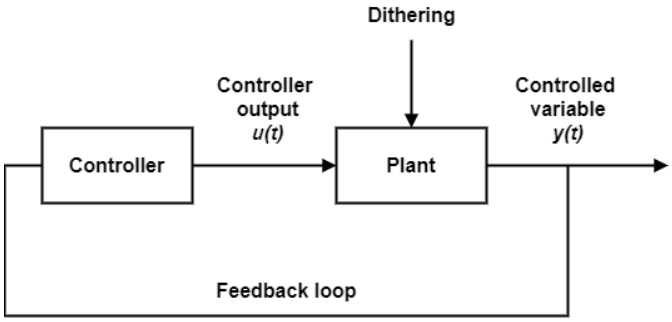

Fig. 3. Controller pattern in Kubernetes

In the state-space model, such a system can be described as:

$$\dot{x} = Ax(t) + Bu(t)$$
$$y(t) = Cx(t) + Du(t), \quad (1)$$

where $x \in \mathbb{R}^{n\times 1}$, $u \in \mathbb{R}^{m\times 1}$ and $y \in \mathbb{R}^{p\times 1}$- the state of the system, input (or control), and output vectors, respectively. Matrix $A \in \mathbb{R}^{n\times n}$- state or system matrix, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{p\times n}$ and $D \in \mathbb{R}^{n\times m}$- input, output, and feedforward matrices, respectively. In Figure 3 case, D is not necessary.

A controller can be used for providing service level agreement (SLA) since it has formal guarantees about quality of operation under the assumption of the operating environment. However, these guarantees must be tested during the design and testing stage in the software development lifecycle.

A system can be modeled as a continuous-time system or as a discrete. Inputs and outputs vectors can be multi-dimensional and contain different elements that correspond to many different input and output factors. The design of such systems starts with defining control goals and inputs that can get into a system. The number of goals depends on a system's output.

Traditionally, control theory is used for physical and nowadays for cyber-physical systems. It has various methods and a big theoretical basis for managing such systems. Many control problems were solved and mathematically proved.

Cloud is a dynamic and uncertain environment. Cloud-native architecture requires adaptivity to deal with changing requirements and uncertainty. Adaptive software should be able to modify its behavior during the runtime without interrupting itself.

This challenge is not new in the industry and academia and idea to take a theoretical basis and methods from control theory. There are some papers [17], [18], and [19] that describe this approach.

From a software engineering point of view, there are several approaches for creating self-adaptive systems with a feedback loop. A well-known example is MAPE-K that was introduced in [20]. Using MAPE-K it is possible to create a computing environment with functionality to adapt to changes in requirements, business processes and manage itself at runtime.

MAPE-K loop includes monitoring managed resources and determines an attribute to analyze. If adaptation is required, a plan function selects procedures to achieve the desired state. After it, these procedures execute action recommended by the plan function. The execution step updates the entire knowledge about the computing environment.

A control theory-based controller development process contains 5 steps: identifying control goals, identifying constraints, define system model, develop controller, testing, and validation.

For software control goals may be functional and non-functional requirements. Constraints may lead from requirements and target environment, e.g. amount of computing resources, time for a spin up a virtual machine. The system model should describe the relationship between control goals and constraints. Based on the model a controller can be designed and implemented. After development, it should be tested and validated. It can be done using DevOps approaches.

In control theory, these steps are clearly defined and have a lot of principles and recommendations. Merging software engineering challenges with control theory can increase the quality of software with mathematically proved methods and laws [21].

VIII. KUBERNETES OPERATOR AS A CONTROL SYSTEM FOR CLOUD-NATIVE APPLICATIONS

Developing the idea behind automatic control and considering the use of the controller pattern in Kubernetes, we can say that a Kubernetes operator is a control system for the Kubernetes application. Moreover, since the cloud environment and cloud-native architecture are based on controller pattern usage it is the best place to implement this idea.

Also, this idea finds confirmation in [3] and [22]. "An adaptive system can be coupled with an adaptation manager to make it continuously satisfy its requirements." [3]. In addition, in the industry Kubernetes operators often have a "manager" word in their names.

Figure 4 shows the idea in the representation from the control theory point of view.

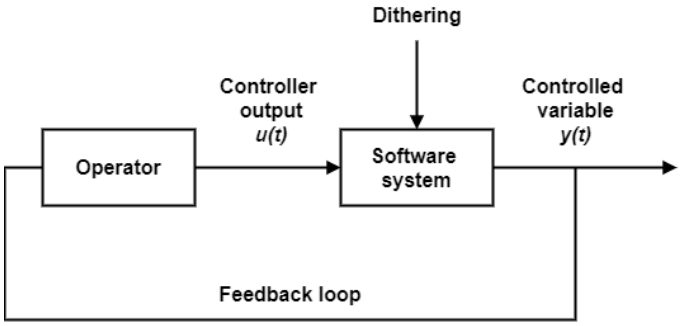

Fig. 4. Operator as a control system

A system can be described as (1). The system with changes on that an operator should react can be modeled as:

$$\begin{cases} \dot{x} = Ax(t) + Bu(t), t < t_c \\ \dot{x} = (A + \Delta A)x(t) + (B + \Delta B)u(t), t \geq t_c \end{cases} \quad (2)$$

and

$$\begin{cases} y(t) = Cx(t), t < t_c \\ y(t) = (C + \Delta C)x(t), t \geq t_c, \end{cases} \quad (3)$$

where t_c is the time instance when a change has occurred. ΔA – matrix that contains a new system state. ΔB and ΔC represent changes in the input and the output matrix, respectively.

An operator is a controller and its development process should contain all 5 steps from the development process of traditional controllers. In applications running in Kubernetes, usually control goals are performance, reliability, and availability. Controlled variables can be the number of instances of an application, network configuration (load balancing), affinity, etc. All this should be considered when developing the operator.

For creating Kubernetes Controller there are 2 the most popular projects – Operator SDK by CoreOS [23] and Kubebuilder [24] by Kubernetes SIGs. These projects make writing operators easier for developers by providing API, abstractions, and tools for code generation for building Kubernetes API and controllers.

Controllers which generated by these projects contains special function for creating reconciliation (feedback) loops:

```
func (…) Reconcile(request reconcile.Request) (…) {
  app:= &v1.App{}
  err:=r.client.Get(context.TODO(),…)
  if err != nil {}
  changed := checkChanges()
    if changed { }
  ...
  return reconcile.Result{…}, nil
}
```

This function contains an implementation of the control loop and automatically registers this loop and controller in Kubernetes API.

Kubernetes itself periodically monitor build-in and custom resources. If a change occurs or the resync period is finished, it sends a request to a resource controller to reconcile this resource. In this function, it checks with the special client if an application instance was changed and performs some actions depending on this change.

Summarizing, it can be said that the Kubernetes ecosystem has the necessary functionality to implement controllers with feedback loops based on methods and laws of control theory.

IX. Applying control theory to kubernetes operators

From Control Theory perspective the plant is a software system controlled by a controller. In the Kubernetes environment, an operator is a control system (controller) that watches for a software system.

For example, the Postgres-operator manages the Postgres database application [25], Confluent Operator deploys the Confluent streaming platform that is based on Kafka [26], Pravega operator creates and controls Pravega streaming storage which is part of enterprise proprietary solution called "Dell EMC Streaming Data Platform" [27].

During the design and verification of control systems, it necessary to check key properties of control systems like stability, controllability, observability, robustness. These properties can be considered from Kubernetes operators perspective too.

Stability is a property of control systems to return to a given or close to its operating mode after any disturbance. Stability has different definitions for both linear and nonlinear systems. Lyapunov stability and some other criteria are used in regular control theory for such cases. In [28] proposed methods to transfer criteria to software. For applications that are managed by operators disturbances can node failure, network glitches, restarts of application instances. For example, the operators that we have given above can cope with such perturbations.

Another important property is controllability. This property shows the ability to transfer the system from one state to another. This is one of the mandatory steps in the synthesis of control systems. To prove controllability, we can use the controllability criteria, which states that a linear system (1) is completely controllable if the rank of the controllability matrix is *n*. For Kubernetes operators, it means that they should be able to move a software system from failed or maintenance state to a normal.

Observability is a property that shows whether it is possible to completely restore information about the states of the system at the exit. It is necessary to have information about the current state of the system *x(t)* at each moment of time. The measurable and observable are output variables *y(t)*. To prove observability, we can use the observability criteria, which states that a linear system (1) is completely observable if the rank of the observability matrix is *n*. Reconciliation loops in operators use this property to achieve control goals. Operators have only output information (current state of the system) and based on this information they perform control actions.

Some examples of models of software and examination of properties listed above are given in [29].

X. Conclusions and future work

This paper describes the cloud-native architecture and cloud-native applications. With gained usage of cloud environments, these concepts will become more popular and important. Also, we described Kubernetes operators for complex stateful cloud-native application as a control system and made mapping between operators concepts and control theory.

Cloud-native architecture is based on principles such as virtualization, service-orientation, uncertainty, adaptivity. Also, it includes patterns such as microservices, models, and usage of controllers with a feedback loop.

Cloud-native applications are containerized, microservice-based, and dynamically managed software. Such applications implement cloud-native applications and are designed to work in a cloud environment.

CNA is mostly managed by Kubernetes. Kubernetes is the most popular container orchestrator. It has an extendible API

with custom resources and custom controllers. Controllers' essence is the feedback (reconciliation) loop. Because of it, the idea to merge software engineering with methods and mathematical laws of control theory become more popular.

Besides this idea, CoreOS developed a new type of software called "Operators". An operator is a controller with specific domain knowledge.

Our next steps continue researching Kubernetes operators and create an example of how to apply control theory, its method, and steps for designing a controller to a complex stateful application that should be launch in Kubernetes.

After that, we want to develop recommendations for developers on how to develop operators better based on control theory laws. Then we want to propose to incorporate these steps and recommendations into Operator SDK and Kubebuilder.

REFERENCES

[1] Linthicum, D. S. (2017). Cloud-Native Applications and Cloud Migration: The Good, the Bad, and the Points Between. IEEE Cloud Computing, 4(5), 12–14.

[2] Arcelli, Davide & Cortellessa, Vittorio. (2016). Challenges in Applying Control Theory to Software Performance Engineering for Adaptive Systems. 35-40. 10.1145/2859889.2859894.

[3] A. Filieri et al., "Software Engineering Meets Control Theory," 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2015, pp. 71-82, doi: 10.1109/SEAMS.2015.12.

[4] S. Shevtsov, M. Berekmeri, D. Weyns and M. Maggio, "Control-Theoretical Software Adaptation: A Systematic Literature Review," in IEEE Transactions on Software Engineering, vol. 44, no. 8, pp. 784-810, 1 Aug. 2018, doi: 10.1109/TSE.2017.2704579.

[5] A. Ahmad, P. Jamshidi, and C. Pahl. 2014. Classification and comparison of architecture evolution reuse knowledge – A systematic review. J. Softw.: Evol. Process 26, 7 (2014), 654–691.

[6] Pahl C., Jamshidi P. (2015) Software Architecture for the Cloud – A Roadmap Towards Control-Theoretic, Model-Based Cloud Architecture. In: Weyns D., Mirandola R., Crnkovic I. (eds) Software Architecture. ECSA 2015. Lecture Notes in Computer Science, vol 9278. Springer, Cham

[7] Pahl, C, Jamshidi, P, Weyns, D. Cloud architecture continuity: Change models and change rules for sustainable cloud software architectures. *J Softw Evol Proc*. 2017

[8] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. 2018. Architectural Principles for Cloud Software. ACM Trans. Internet Technol. 18, 2, Article 17 (February 2018), 23 pages.

[9] CNCF, "cncf/toc," GitHub. [Online]. Available: https://github.com/cncf/toc/blob/master/DEFINITION.md. [Accessed: 11-Jan-2021].

[10] N. Kratzke and R. Peinl, "ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects," 2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW), Vienna, 2016, pp. 1-10.

[11] E. Carter, "Sysdig 2019 Container Usage Report: New Kubernetes and security insights," Sysdig, 29-Oct-2019. [Online]. Available: https://sysdig.com/blog/sysdig-2019-container-usage-report/. [Accessed: 6-May-2021].

[12] B. Burns, Kubernetes - Up and Running: Dive into the Future of Infrastructure. OReilly Media, Incorporated, 2019.

[13] "Kubernetes Components," Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Accessed: 12-Feb-2021].

[14] "Extending your Kubernetes Cluster," Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/extend-cluster/. [Accessed: 12-Feb-2021].

[15] "Introducing Operators: Putting Operational Knowledge into Software," CoreOS. [Online]. Available: https://coreos.com/blog/introducing-operators.html. [Accessed: 3-March-2021].

[16] "kube-controller-manager," Kubernetes. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/. [Accessed: 3-March-2021].

[17] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M.Litoiu, H. Muller, M. Pezze, and M. Shaw. "Engineering Self-Adaptive

[18] Systems through Feedback Loops". In: Software Engineering for SelfAdaptive Systems. Vol. 5525. LNCS. Springer, 2009, pp. 48–70.

[19] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. "Reliability-driven dynamic binding via feedback control". In: SEAMS. 2012, pp. 43–52

[20] A. Filieri, H. Hoffmann, and M. Maggio. "Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees". In: ICSE. ACM, 2014, pp. 299–310.

[21] J. Kephart and D. Chess. "The vision of autonomic computing". In: Computer 36.1 (2003), pp. 41–50

[22] Pahl C., Jamshidi P. (2015) Software Architecture for the Cloud – A Roadmap Towards Control-Theoretic, Model-Based Cloud Architecture. In: Weyns D., Mirandola R., Crnkovic I. (eds) Software Architecture. ECSA 2015. Lecture Notes in Computer Science, vol 9278. Springer, Cham

[23] "Operator-Framework/Operator-SDK," GitHub. [Online]. Available: https://github.com/operator-framework/operator-sdk/. [Accessed: 16-March-2021].

[24] "kubebuilder," GitHub. [Online]. Available: https://github.com/kubernetes-sigs/kubebuilder/. [Accessed: 16-March-2021].

[25] "zalando/postgres-operator", GitHub, 2021. [Online]. Available: https://github.com/zalando/postgres-operator. [Accessed: 04- May-2021].

[26] "Confluent Operator | Confluent Documentation", Docs.confluent.io, 2021. [Online]. Available: https://docs.confluent.io/operator/current/overview.html. [Accessed: 04- May- 2021].

[27] "pravega/pravega-operator", GitHub, 2021. [Online]. Available: https://github.com/pravega/pravega-operator. [Accessed: 05- May-2021].

[28] M. Roozbehani, A. Megretski and E. Feron, "Optimization of Lyapunov Invariants in Verification of Software Systems," in IEEE Transactions on Automatic Control, vol. 58, no. 3, pp. 696-711, March 2013, doi: 10.1109/TAC.2013.2241472.

[29] Abdelzaher T., Diao Y., Hellerstein J.L., Lu C., Zhu X. (2008) Introduction to Control Theory And Its Application to Computing Systems. In: Liu Z., Xia C.H. (eds) Performance Modeling and Engineering. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-79361-0_7