Your Very Nice Project Title

Mark Zhang, William Zheng
University of Illinois at Urbana-Champaign
{zz91,xinzez2}@illinois.edu

Abstract

Cloud application systems running on top of by container orchestration systems such as Kubernetes are often managed by domain-specific controller code called operator to automate laborious manually-done operations. The correctness and the reliability of such code is critical to productionized application yet operators are highly complex and difficult to ensure correctness. We conduct empirical study on 52 popular Kubernetes operator from a software engineering perspective to characterize their complexity. TODO

1 Introduction

With the prevalence of cloud-native software systems, Kubernetes [1] has become one of the popular choices for orchestrating productionized container applications. Developers extend Kubernetes APIs and functionalities by writing specialized programs called operators, which automate the laborious task of deploying and managing services and applications. Since operators are responsible for various highly complex operations such as service scaling, backup, and migration in production environment, their correctness is paramount for system reliability. To understand the reliability of operators and how it could be improved, we must first develop a deep understanding of the complexity of operators and operators fail.

The significance of reliability of Kubernetes operators stems from challenges inherent in their design and execution within production environments. Operators undertake diverse and mission-critical responsibilities, including but not limited to dynamic scaling of services, data backup, and seamless application migration. As a result, any failures within operator functionalities can cause cascading repercussions, ranging from service disruptions to potential data loss, ultimately undermining the stability and availability of the entire system.

2 Methods

To understand the reliability implications in Kubernetes operators, we collect various metrics from 52 popular open-sourced operators. Each of the operators is then analyzed to understand the complexity of the operators and the challenges faced by the developers in writing reliable operators.

2.1 Operator Selection

The 52 operators are selected by students in the UIUC Spring 2024 CS523 Advanced Operating Systems course. Each student is instructed to choose one operators that they are interested in and to ensure the operators are popular, mature, and well-maintained. Popularity and maturity are measured by the number of stars and forks on GitHub, and the maintenance status is measured by the last commit date.

The operators are selected from a variety of domains, including but not limited to message queue, database system, and distributed synchronization systems.

The operators selection might not be representative of the entire Kubernetes operator ecosystem. The selection might be biased towards the students' interest and the managed system they are familiar with.

2.2 Data Collection

We collect the following metrics from the operators:

- **LOC and programming language distribution**: We collect the number of lines of code (LOC) and the programming language used in writing the operators. We use the LOC as a proxy for the complexity of the operators.
- **Operator developers' involvement in the managed system development**: We collect the involvement of operator developers in contributing to the managed system using GitHub repository contributor data. We use this metric to understand the familiarity of operator developers with the managed system.

3 Results

3.1 Operator Code Complexity

Lines of code in a software project is one indicator of system complexity. We counted the lines of code (LOC) each Kubernetes operator codebase by the language used. Figure 2 shows that the total LOC follows a power law distribution. The largest codebase, azure-service-operator, has 4.3 million LOC, while the smallest codebase, container-security-operator, has 7755 LOC. Most operators are written in Go while 3 are written in Java, and 1 written in Python. On average, 44.5% of the total lines of code in the project is the programming language that the operator is written in; 3

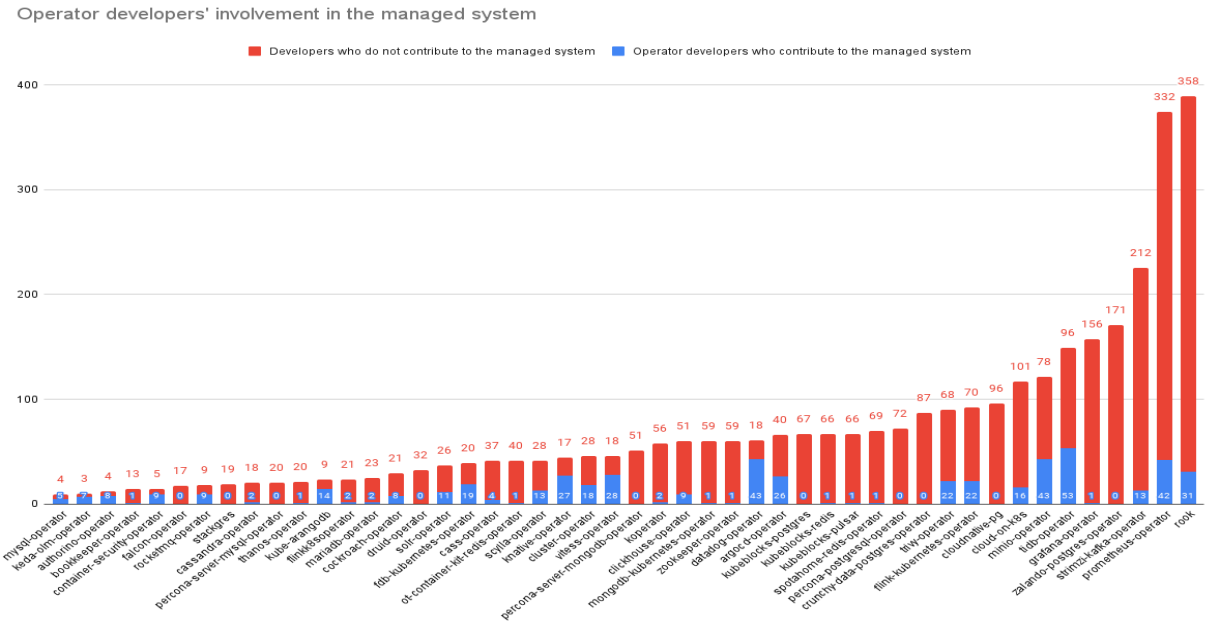

Figure 1. Operator developers' involvement in the managed system developement

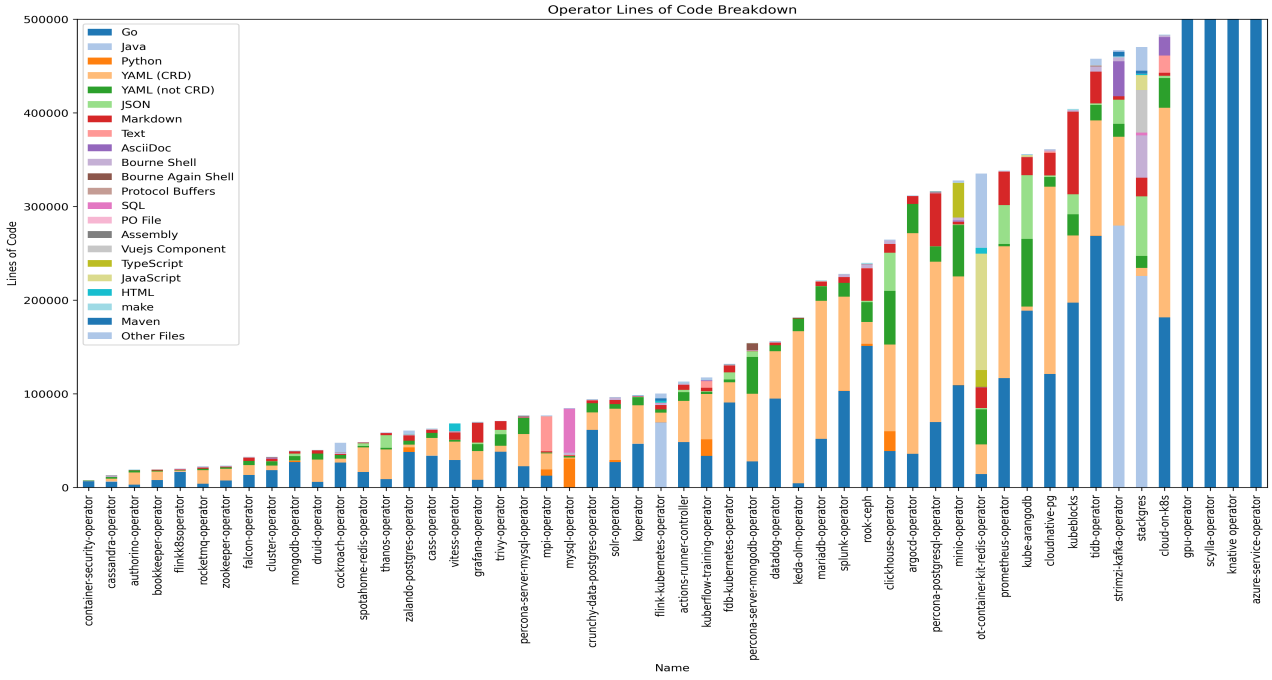

Figure 2. LOC and programming language distribution over operators

3.2 What are the developers' challenges?

Operators are responsible for operating a very specifc managed system and it is operators' job to reconcile the managed system to the desired state given users' expectation. Modern software systems can be gigantuous with tens of thousands lines of code and complicated states management required. This engenders burdens for operator developers to understand the managed system thoroughly, increasing

the likelyhood for them to make mistakes in developing the operator.

In Figure 1, we collect and visualize the involvement of operator developers in contributing to the managed system using GitHub repository contributor data. While the absence of operator develoipers' contribution to the managed system does not necessarily imply their incompetence in understanding the managed system, it is a still a reflective metric on the average familiarity of operator developers on the managed system. In our dataset, only 19% of the operator projects have more than half of the developers being contributing to the managed system. The proportion of developers who are also developing the managed system drops as the operator project gets larger. In the two largest operator project in our dataset, with regard to the size of developers, only 10% of the developers are contributors to both the managed system and its operator.

3.3 What are the testing practices of Operators?

Operators are responsible for managing the state of the managed system. To ensure the correctness of the operators, developers write tests to verify the correctness of the operators. We analyze the testing practices of the operators by counting the number of tests in the operators.

TODO: Add more data and analysis here.

Operator	Project Stars	Test Cases
Strimzi-Kafka-Operator	4.3K	1676
Percona-Xtradb	489	130
CloudNative-pg	3K	1426
TiDB-Operator	1.2K	103
MinIO-Operator	1.1K	80

Table 1. Automated Test Breakdown by Operator

4 Related Work

There are previous efforts in understanding device driver in modern system[2]. The operator shares a very similar figure with the devce driver. Both of them impose the challenges of reliably bridging the gap between managed system and the underlying operating system interfaces, while the underlying system for device driver is operating system and the underlying system for the operator is Kubernetes. Though both modern operating system and cloud management system are complicated, interacting with cloud management system faces even more problems in handling distributed asynchronous issues. The cloud environment raises new classes of reliability challenges like staleness and faults when interacting with cloud management system[3] ,which requires extra attention from operator developers to avoid bugs. This work focuses on both common reliability problem faced by operator and device driver, and the newly raised reliability problem in the cloud environment.

5 Conclusion

This project is awesome.

6 Metadata

The presentation of the project can be found at:

https://zoom/cloud/link/

The code/data of the project can be found at:

https://github.com/you/repo

References

[1] Kubernetes. https://kubernetes.io/, 2024.

[2] Kadav, A., and Swift, M. M. Understanding modern device drivers. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, Association for Computing Machinery, p. 87–98.

[3] Sun, X., Luo, W., Gu, J. T., Ganesan, A., Alagappan, R., Gasch, M., Suresh, L., and Xu, T. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 143–159.