Migrating the Openshift CI infrastructure to a microservice architecture

Michail Kargakis
Red Hat, Inc.
mkargaki@redhat.com

Abstract

In order to adapt in a demanding and fast-paced release cadence that Openshift follows as a productized distribution of Kubernetes, one of the things we had to rethink as members of the Openshift Continuous Delivery team over the past months was how to scale our CI infrastructure to quickly and efficiently verify, test, and merge developer changes across multiple Github repositories, that eventually get delivered to our customers. Luckily for us, this scaling problem has been already taken up by the *testing* special-interest group[1] in the Kubernetes community so we ended up joining forces in order to push forward the current state of things. The present essay is meant to serve as a brief description of our journey from a monolithic to a microservice CI architecture, an introduction to the new CI system that is based on the concept of microservices, and why we are better off in this new world for our CI infrastructure.

1 Introduction

Google released Kubernetes during the summer of 2014, in an effort to bring its experience in managing workloads with Borg into the open-source container world that had started booming with the uptake of technologies like Docker. Red Hat, among a slew of other companies, small startups and large enterprises alike, rallied behind Kubernetes, resulting in building a great community that is the primary factor for the success of the project nowadays. Kubernetes follows a short release cycle, adhering to the *release early, release often* philosophy and has ended up being one of the most active Github projects with a total of ~1400 unique contributors[2], ~200 of them being active every month as of the time of this writing. In order to meet the project's needs, the community had to come up with a solution for scaling the CI infrastructure to effectively test and merge all incoming contributions in a timely fashion and at the same time boost developer productivity. *prow*[3] is the result of this effort.

2 Background

Back in the early days of Kubernetes, the volume size of pull requests that landed in Openshift was trivial enough that a single tool could effectively handle both testing and merging. We had been using Jenkins for test execution and a ~2,5kloc monolithic Ruby bot[4] that was responsible for triggering jobs in Jenkins in response to Github comments and merging pull requests with successful tests. As the number of contributions grew, we started experiencing issues with the merge ratio in our main repository. Failures in tests unrelated to the changes introduced by a pull request (also known as *flakes*) would cause the pull request to be re-queued at the end of a slow serial merge queue. It became obvious that we needed to improve the state of our CI tooling[5]. The first course of action was to enhance our bot. *prow* was already thriving in the Kubernetes community and one of its killer features, batch merging, enabled upstream contributors to iterate quickly on a high volume of pull requests. Adding batch merging in our bot didn't turn out to be a simple task. Actually, due to the monolithic and far from modular nature of the tool, that option was quickly abandoned. At the same time, we evaluated replacing our bot with *prow*. It took some changes upstream to ensure that *prow* could be configured to work for projects outside the Kubernetes ecosystem and its seamless integration with Jenkins is what enabled us to eventually switch to it.

3 CI choreography

prow can be described as a CI orchestrator. It only integrates with Github today and supports a wide variety of test types. It can run tests for pull requests (presubmit), tests for merged commits (postsubmit), and periodic tests not necessarily related to Github repositories. There is also support for various plugins that are triggered on different Github events in order to enhance contributor experience. Looking into its internals, *prow* is a collection of small, self-contained, battle-tested micro-services written in Go. There are clear boundaries and no dependencies between services. Following the Unix philosophy, each service does one thing and does it well. All of the services have been built to run on top of a Kubernetes cluster which also serves for dogfooding the work that is being done in the Kubernetes community.

The base unit of *prow* is a *ProwJob*, a representation of a test run serialized in a Kubernetes custom resource[6] and stored by the Kubernetes API server[7]. *ProwJobs* and the Kubernetes API are what actually enables all *prow* components to work together, without the need to communicate directly, in a service choreography[8].

All services use the Kubernetes API and crud[9] *ProwJobs*. In total there are seven different services that comprise *prow.* One service is responsible for listening on Github events and reacting based on the event type. One common action it takes is to create *ProwJobs*, essentially the tests that need to run, whenever a pull request by a trusted organization member is opened on Github. Different services are responsible for managing the lifecycle of all *ProwJobs* in the system by executing the actual tests in different agents. Today, tests can run either in Jenkins, or in Kubernetes pods[10] but *prow* is extensible enough that it is a matter of adding a new service that handles the logic of starting jobs in a new agent in order to integrate that agent with *prow*. Garbage-collection of *ProwJobs*, creation of periodic jobs based on intervals, merging Github pull requests, and a frontend to *prow* that provides a view to all *ProwJobs*, all are functions handled by different services, too.

4 Conclusion

There is still work that needs to be done in order to stabilize the *ProwJob* API and make *prow* more operation-friendly. Some examples include having documentation for debugging flows and exposure of metrics so we can quickly take actions when required. All that being said, *prow* already brings many benefits to the table. It is easier to introduce a new change due to the extensibility of the system, and the nature of the code – self-contained, small microservices written in Go – boosts its debuggability (components can be tested independently in the same fashion they would be tested as a whole), maintainability (small codebase plus most developers in the community are familiar with Go since it's the main language used both in Openshift and Kubernetes), and confidence on the project as a whole (flashing startup times, zero to *prow* deployments are bound by build times and even these are short since we are building small binaries).

5 Acknowledgments

I would like to take the opportunity and give credits to Joe Finney and the rest of the sig-testing and sig-contribex folks who have helped give *prow* life and shape. Steve Kuznetsov who is my main partner in crime for driving this transformation in our CI infrastructure. Paul Weil and Jeff Vance for reviewing this article, and all of the contributors, either Red Hatters or other folks in the Kubernetes community, who have submitted constructive feedback in Github issues and personal discussions.

References

[1] https://github.com/kubernetes/community/tree/master/sig-testing - Mostly Google engineers at this point since all of the CI infrastructure is currently provided by Google. There is an effort in moving parts of it under the purview of CNCF, the home organization of Kubernetes ever since Google donated the project.

[2] https://github.com/kubernetes/kubernetes/graphs/contributors

[3] https://github.com/kubernetes/test-infra/tree/master/prow

[4] https://github.com/openshift/test-pull-requests

[5] Test flakes are usually deficiencies either in production code or in the actual tests (or network hiccups if the tests depend on networking). Optimizing around them is a bad thing but it wasn't the only reason for pushing forward for better tooling.

[6] https://kubernetes.io/docs/concepts/api-extension/custom-resources/

[7] All Kubernetes resources are actually stored in etcd, a key-value storage that is managed by the Kubernetes API server.

[8] https://en.wikipedia.org/wiki/Service_choreography

[9] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[10] https://kubernetes.io/docs/concepts/workloads/pods/pod/