What goes wrong in serverless runtimes? A survey of bugs in Knative Serving

Tim Goodwin
University of California, Santa Cruz
USA
timgoodwin@ucsc.edu

Andrew Quinn
University of California, Santa Cruz
USA
aquinn1@ucsc.edu

Lindsey Kuper
University of California, Santa Cruz
USA
lkuper@ucsc.edu

ABSTRACT

Serverless runtime systems are complex software artifacts and difficult to make reliable. We present a large-scale empirical study of bugs in serverless runtimes, in the context of the popular open-source Knative Serving serverless platform. We analyze issues reported against Knative Serving over a three-year period and identify broad trends. Our findings shed light on the challenges of building correct, efficient serverless runtimes and suggest fruitful directions for further research.

ACM Reference Format:
Tim Goodwin, Andrew Quinn, and Lindsey Kuper. 2023. What goes wrong in serverless runtimes? A survey of bugs in Knative Serving. In *The 1st Workshop on SErverless Systems, Applications and MEthodologies (SESAME '23), May 8, 2023, Rome, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3592533.3592806

1 INTRODUCTION

Serverless computing promises to drastically simplify distributed system deployments by automatically deploying and managing the compute infrastructure underlying an application's code. In particular, the approach automatically deploys more compute resources when system load increases and removes compute resources when system load decreases, potentially even decommissioning all resources and "scaling-to-zero". Thus, in the ideal case, serverless computing achieves perfect utilization: every deployed resource performs useful work for the application, and when there is no work to be done, the application can remain available without any backing resources.

Unfortunately, the ideal serverless vision is elusive since it requires that a *serverless runtime* (i.e., the system that manages resources for a serverless application) attain complex, and sometimes even contradictory, design goals.

For example, serverless runtimes aim to maintain application availability while supporting dynamic changes to an application's configuration and its load. Autoscaling can help attain this goal, yet a runtime's autoscaling must balance utilization and performance.

For utilization, a runtime decommissions resources that are not in use by an application to reduce waste. However, decommissioning resources can lead to the notorious "cold-start" problem, in which subsequent invocations require applications to execute on recently deployed resources that are "cold" (i.e., performance-critical system caches are empty and code must be loaded from storage into memory) [Mohan et al. 2019]. These challenges are made even more complex by emergent properties that arise when seemingly independent design goals interact.

In this paper, we empirically study the key challenges faced when building a correct and efficient serverless runtime. We first present the first large scale study of bugs in serverless runtimes. In particular, we study all the 103 reported bugs that cause an issue in runtime behavior from the last three years in Knative [Knative Authors 2023], a popular open-source serverless platform. We choose Knative because: (1) Knative is open-source and has a public issue tracker; (2) Knative is in active development, with new features and bug fixes released each day; (3) Knative is mature, having been in development since July 2018; (4) Knative is integrated into Kubernetes [Kubernetes Authors 2023a], the industry standard container orchestration platform; and (5) Knative is an enterprise-level solution, with adoption from major companies including Google, VMWare and IBM. Section 2 provides additional details on Knative including its design and integration into Kubernetes.

We then taxonomize the bugs in our study into seven overlapping categories based upon the design goals that they violate as well as the system component involved in the bug. Our taxonomy reveals that the majority of bugs fall into one (or more) of four categories, with each of the four categories accounting for at least 25% of all bugs: (1) *status condition* bugs, i.e., bugs relating to how Knative determines the health and readiness of its components; (2) *Kubernetes interaction* bugs, i.e., bugs relating to how Knative interacts with Kubernetes; (3) *configuration* bugs, i.e., bugs in which Knative does not correctly support a configuration parameter or combination of parameters; and (4) *autoscaling* bugs, i.e., bugs in which Knative's Autoscaler does not work correctly. Section 3 details our study methodology and bug taxonomy, and provides examples for each category.

We identify three key takeaways from our survey and propose three corresponding directions for future work (Section 5). First, we observe that Knative's intended semantics are often poorly understood, even by Knative developers. Thus, we suggest formalizing a more accurate and precise serverless runtime semantics. Second, Knative developers often lack an understanding of how code changes will interact with different parts of the system, as shown by the high number of Kubernetes interaction bugs. Thus, we suggest investigating automated testing for Knative. Finally, we observe that while Knative configuration bugs are *not* misconfiguration bugs, configuration bugs are often "fixed" by changing the value of default parameters. Thus, we suggest work to apply automated

SESAME '23, May 8, 2023, Rome, Italy

ACM ISBN 979-8-4007-0185-6/23/05.
https://doi.org/10.1145/3592533.3592806

testing, automated configuration checking tools, or formal methods to configurations.

In summary, we make the following contributions:

- We perform the first large-scale study of bugs in serverless runtimes.
- We taxonomize serverless runtime bugs into seven groups and identify broad trends.
- Based on our analysis of bugs, we identify promising areas of future work on serverless runtimes.

2 A KNATIVE AND KUBERNETES PRIMER

In this section, we provide background on Knative and Kubernetes. We study Knative due in large part to its popularity in the computing industry. Moreover, we observe that Knative is highly flexible and extensible; it avoids locking its users into any particular design trade-off [Kaviani et al. 2019]. As a result, whereas a highly tailored serverless runtime might only illuminate a few interesting design tradeoffs faced by a serverless runtime, Knative's generality means that it explores a significantly larger design space. In sum: we observe that Knative is perhaps an ideal single system to study in order to understand the set of issues that will arise when building a serverless runtime.

In the remainder of this section, we describe Kubernetes (Section 2.1), with particular emphasis on the concepts and patterns that Knative inherits from Kubernetes. Then, we describe the core components of Knative and how Knative extends Kubernetes to provide a serverless programming model and runtime (Section 2.2).

2.1 Relevant Kubernetes Concepts

Kubernetes is the container orchestration platform underlying the Knative serverless platform. In Kubernetes, applications are deployed using a combination of *resources*, which are abstract representations of computing components that can be created, managed, and deleted through the Kubernetes API. Resources are defined via manifest files containing attributes which specify their behavior and characteristics. Knative relies upon three Kubernetes resources — pods, deployments, and services — to deploy and invoke functions:

- *pod*: The smallest deployable unit of compute; a thin wrapper around one or more containers.
- *deployment*: A resource that identifies how to create or modify pods that run an application.
- *service*: A collection of pods running the same application that can be addressed through the same network endpoint.

Kubernetes manages resources with *controllers*, which are routines that implement a control loop to manage a resource's desired state. The control loop continuously monitors the current state of the resource, compares it to the desired state specified in a Kubernetes manifest, and takes corrective action to bring the system back into the desired state as necessary. In addition to configuration values provided by the user, a resource's desired state may include values maintained dynamically by the system. For example, the deployment controller maintains the desired number of pods in a deployment and will create replacement pods in response to a pod failure. The deployment controller may also create or remove pods if an autoscaling policy causes the deployment's desired number of pods to change.

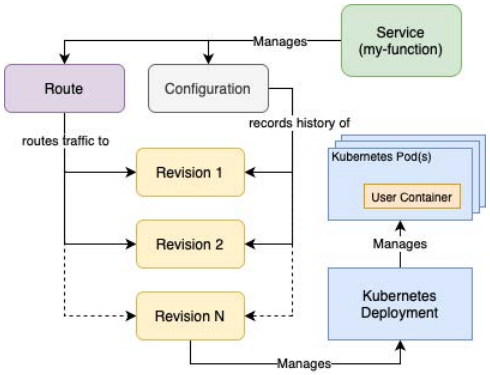

Figure 1: The primary resources involved with deploying a container application on Knative Serving

2.2 Knative Components

The Knative project consists of three subcomponents: Knative Serving, Knative Functions, and Knative Eventing. Knative Serving represents Knative's serverless runtime; it defines and controls how serverless workloads are deployed on Kubernetes and contains the infrastructure that supports Knative's autoscaling and scale-to-zero functionality. Knative Functions provides the programming model that allows users to write stateless functions for Knative without knowledge of Kubernetes, the Knative runtime, or even containers. Knative Eventing exposes a collection of APIs for building event-driven applications with Knative.

The Knative Functions Command Line Interface packages a user's serverless "function" into a container environment. Knative Serving then uses this image and an associated configuration to generate the resources that underlie a Knative service.

The three primary resources involved with deploying an application on Knative Serving are *configurations*, *revisions*, and *routes*. A *configuration* specifies the desired state of the Knative application and manages the creation of new revisions. A *revision* is an immutable snapshot of a container application together with its configuration; whenever a user modifies an application's code or updates its configuration, Knative creates a new revision. Finally, *routes* map a network endpoint to one or more revisions.

Together, these resources implement a network service on Kubernetes. Knative orchestrates these resources with a dedicated controller for each resource. For example, when a user deploys a Knative function, the configuration controller creates a revision. The revision controller then creates a Kubernetes deployment, which ultimately runs the application's container instances.

Knative coordinates these resources to support various deployment strategies such as enabling progressive roll-out and roll-back of application changes. Knative can shift application traffic to a new revision gradually, and can replace a buggy revision with the previous version if issues arise during deployment. The relationship between these resources is depicted in Figure 1.

Conditions. Knative provides high-level status reporting on the health and readiness of its resources by extending the Kubernetes pattern of *conditions* [Kubernetes Authors 2023b]. Knative Serving

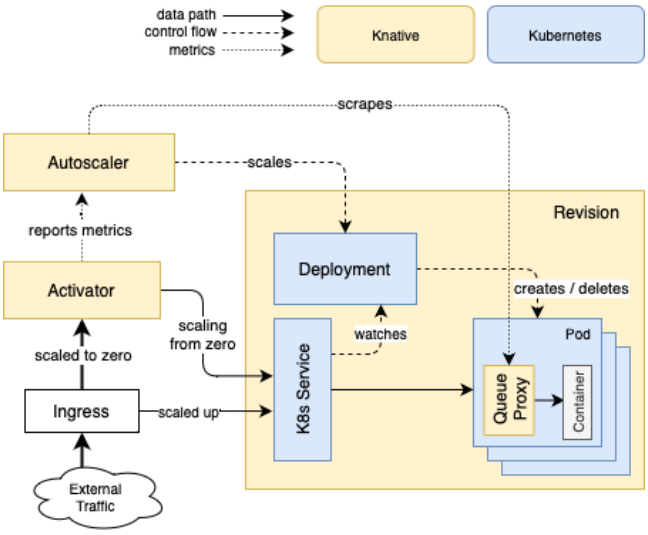

Figure 2: A high-level depiction of the data paths and components involved in scaling a revision's instances from zero.

implements a number of condition types, including Ready, Active, Container Initialized, etc. Knative controllers read and update conditions as the state of the system changes; as follows, status conditions serve as a means of communication between controller control loops. Namely, in each control loop, a resource's controller will observe the state of its dependencies by reading their status conditions. A controller may then use these observations to update the status condition values for the resource it manages.

Autoscaling. A key feature of Knative Serving is its autoscaling and scale-to-zero functionality. The runtime automatically manages the number of pods in a revision to account for changes in the request load in the system. The key components include:

- Autoscaler: A component that updates the number of pods in a revision's deployment based upon traffic metrics, which then triggers the Kubernetes's deployment controller to modify the number of pods that are deployed.
- Activator: The component that forwards traffic metrics to the Autoscaler. Additionally, when a revision has no pods, the activator buffers any requests that arrive to that revision until the Kubernetes's deployment controller has created new pods.
- Ingress Gateway: A component that manages traffic routing and network programming to make Knative services accessible from the outside world.

Figure 2 depicts how these components work together to implement Knative's autoscaling and scale-to-zero functionality.

3 WHAT ARE THE BUGS?

We next present our bug survey. We survey *all* the runtime-behavior bugs in Knative's serverless runtime that were reported on the Knative Serving public issue tracker between January 2020 and February 2023, resulting in a dataset of 103 bugs.[1]

Category Name	# Tagged	% Resolved	Example Issues
status conditions	40	74%	#10267, #8539
Kubernetes interactions	32	78%	#13204, #12538
configuration	28	85%	#11926
autoscaling	27	81%	#8610, #8685
request routing	18	66%	#12593, #11532
version semantics	7	71%	#12538
invocation history	5	60%	#6504

Table 1: The number of bugs we identified in each category, and a sampling of representative issues from each category.

We taxonomize the bugs into seven overlapping categories, based upon the design principle that the bug violates, the component that the bug involves, or the bug's root cause. Table 1 identifies the number of bugs and a list of representative issues for each category; a single issue may be resident in multiple categories. In brief, the categories are:

- *status conditions* (Section 3.2.1): issues relating to how Knative determines the health and readiness of its components.
- *Kubernetes interactions* (Section 3.2.2): issues relating to the interaction of controller processes within Knative and Kubernetes.
- *configuration* (Section 3.2.3): issues relating to how the platform responds to configuration parameters.
- *autoscaling* (Section 3.2.4): issues relating to Knative's autoscaling functionality, including scale-to-zero.
- *request routing* (Section 3.2.5): issues relating to how incoming requests are distributed to available function instances.
- *version semantics* (Section 3.2.6): issues relating to how the platform manages function versions and their updating.
- *invocation history* (Section 3.2.7): issues where the platform handled a function invocation differently based on whether it had been invoked before or not.

In the rest of this section, we first describe the methodology that we followed to gather our dataset (Section 3.1), and then describe the bug categories in detail (Section 3.2).

3.1 Methodology

We survey *all* the publicly reported runtime-behavior bugs in Knative's serverless runtime between January 2020 and February 2023. To limit the survey to issues in the serverless runtime, we limited the survey to issues in the public GitHub issue tracker for Knative Serving[2] (2063 issues). To limit the survey to only bugs, we further filtered issues to only select those that were tagged with the "kind/bug" label (482 issues remaining). To limit the survey to only include bugs in runtime behavior, we then filtered out issues that were tagged with "area/build" or "area/test-and-release" (418 issues remaining), or those containing the keywords "install", "registry", "test", "metrics", "prometheus", or "grafana" in the issue title (325 issues remaining). Finally, we manually analyzed the remaining issues and further filtered out 18 bugs that did not include enough

[1] Our dataset is publicly available at: https://github.com/lsd-ucsc/knative-runtime-bugs
[2] https://github.com/knative/serving/issues

discussion to understand and 204 bugs that did not relate to runtime behavior and were therefore out of scope for our survey (most of these out-of-scope bugs involved user error in configuring or operating Knative). After all of our filtering, we were left with 103 issues that we believed to be genuine bugs in Knative that involved Knative's runtime behavior.

Threats to Validity. Our survey has the following threats to validity. First, our survey relies on a public bug tracker, and is thus inherently incomplete in that not all bugs in Knative's serverless runtime will result in issues posted on Knative's GitHub. Similarly, our survey is incomplete since there may be runtime behavior problems that were reported in the wrong project (e.g., reported in Knative Functions), reported without the "kind/bug" label, incorrectly tagged (e.g., tagged as "area/build"), or contained keywords that indicated the wrong type of behavior (e.g., included the keyword "install"). Our survey is also incomplete since there were some bugs that contained insufficient discussion and thus we had to elide. Finally, while Knative is an interesting and popular serverless runtime, it represents only a single point in the possible design space of serverless runtimes. Thus, while we believe that many trends in our survey would extend to other serverless runtime designs, we cannot offer any assurance that our findings generalize beyond Knative.

3.2 Problem Areas in Runtime Behavior

In the remainder of this section, we discuss each bug category, or *problem area*, that we observed in our survey, in the order of most prevalent to least prevalent.

3.2.1 Status Condition Bugs. The plurality of bugs in our survey are related to how Knative uses status conditions to represent the readiness of its resources. As discussed in Section 2, status conditions indicate the health status and readiness of Knative resources to other Knative controllers and operators.

Status condition bugs arise when controllers contain errors in how they read and update conditions to coordinate resources. An insidious class of status condition bugs arise due to race conditions in how (potentially multiple) controllers interact with (potentially multiple) status conditions. Namely, Knative's use of control loops results in status conditions that do not update immediately. Rather, they are reconciled eventually, as controllers act upon them. So, when multiple controllers update status conditions based on the values of other conditions, the controllers may encounter race conditions, many of which involve reading a status condition that has not yet updated to reflect a change in the component state it represents.

These bugs reveal that although status conditions provide a flexible way to represent the status of a resource, using them to coordinate system components in the dynamic environment of a serverless runtime requires careful handling of their semantics and how the dependencies between them are managed.

Example. Issue #8539 describes a race condition across status conditions that resulted in a revision failing to deploy. In particular, the bug involves a race condition between a revision's Ready status condition and its Pod Autoscaler's Ready status condition (each revision includes a Pod Autoscaler to interact with the Knative Autoscaler). The Pod Autoscaler's Ready status condition is based upon the revision's Active status condition, which signals the existence of active revision pods. In turn, the revision's Ready status condition depends on the Pod Autoscaler's Ready status condition being set to true. Thus, for a revision's Ready status condition to be set, it would need to have already deployed pods. The bug was fixed by removing the dependency between the Pod Autoscaler's Ready status condition and the revision's Active status condition.

```
    ...,
    StartupProbe:             nil,
    Lifecycle:                nil,
-   TerminationMessagePath:   "/dev/termination-log",
+   TerminationMessagePath:   "",
-   TerminationMessagePolicy: "File",
+   TerminationMessagePolicy: "",
-   ImagePullPolicy:          "IfNotPresent",
+   ImagePullPolicy:          "",
    ...
```

Figure 3: Log excerpt from a meaningless deployment update. The revision controller replaces default values set by Kubernetes with empty values because Knative does not specify them in the resource's manifest.

3.2.2 Kubernetes Interaction Bugs. Knative components are tightly integrated into Kubernetes (see Section 2), so controllers from both systems must work together to orchestrate the resources required for function deployments and invocations. A large class of bugs arises from issues involving unintentional interference between Knative controllers and Kubernetes controllers. One common refrain in Kubernetes interaction bugs involves a Knative controller interfering with a Kubernetes controller because it misunderstands the nature of a Kubernetes state change. Another common issue involves Knative controllers and Kubernetes controllers interfering because the controllers respond to the same state change without a mutual awareness of their respective reconciliation efforts. Kubernetes interaction bugs often lead to unavailability, lead to excess resource utilization, or cause the system to enter an incorrect stable state.

Example. Issue #13204 describes a case of excess resource utilization because a Knative controller did not understand the behavior of an existing Kubernetes controller. In this bug, a Knative revision controller created a deployment without specifying all available configuration parameters. Then, a Kubernetes controller set default values for these omitted parameters. The Knative revision controller observed a difference between the deployment's intended configuration and the active configuration (depicted in Figure 3) and issued an update command to the Kubernetes API to reconcile the difference. This interaction would occur when Knative would reconcile all revisions, for example due to the revision controller restarting. The ensuing chain of deployment updates put unnecessary strain on the Kubernetes API server and wastes Knative's API quota (update commands are rate limited). The issue highlights the difficulty in reasoning about which of the many possible state changes a controller must react to, and those which it should ignore. The bug has yet to be fixed as of the writing of this paper.

3.2.3 Configuration Bugs. Knative exposes many configuration parameters so that users can optimize availability, performance, and resource usage. Many bugs we observed (27%) involve Knative Serving incorrectly implementing a configuration setting or behaving incorrectly when configuration values interfere with each other. By "configuration bugs" we are not referring to misconfiguration of Knative *by users*; rather, we are referring to situations when users chose a combination of configuration values that *should* have worked correctly, but did not, due to a bug in Knative Serving itself. Configuration bugs often result in suboptimal resource utilization and performance degradation.

Example. Issue #11926 describes a configuration setup in which the system's steady state was at an inflection point such that the system repeatedly toggled a performance optimization; the frequent churn degraded performance more than just leaving the performance optimization permanently unused would have. In particular, a user deployed Knative with the minimum pods per function set to 2, a request capacity per function pod of 100 (the default), and a target burst capacity of 200 (the default). In periods of high load, Knative improves system performance by moving the Activator component (Figure 2) out of the request processing data-path. Knative initiates the optimization when a revision's remaining request capacity (i.e., its total capacity minus its current load) dips below the target burst capacity. In the user's deployment, the *total* revision capacity was 200 (two revisions with a request capacity of 100 per revision), as was the default target burst capacity, so Knative toggled its faster data-path optimization on and off for each request. Knative developers resolved the issue by modifying the default target burst capacity to 211, a prime number, to make it less likely that a revision's total capacity would collide with the default target burst capacity. This does not solve the oscillation issue, but it does make the problem less likely since it will not occur on the default configuration.

3.2.4 Autoscaling Bugs. Roughly a quarter of all bugs in our survey involve issues with how Knative dynamically manages function resources in accordance with request load. Autoscaling bugs arise when Knative does not scale resources up as load increases, leading to unavailability and poor performance, or does not scale resources down as load decreases, leading to excess resource consumption.

Examples. Issue #8610 describes a scenario where a network outage caused a revision to scale down below what was required for the current load. As described in Section 2, Knative's Autoscaler receives metrics from each revision's activator when determining how many pods to deploy. In this bug, a transient network issue caused the Autoscaler to stop receiving activator metrics; the Autoscaler incorrectly interpreted the lack of messages as a period of no traffic. Consequently, the Autoscaler scaled the revision below its target capacity.

Issue #8685 describes excess utilization because a single request handled by a single pod prevented other pods from being scaled down. To limit request failures when scaling down a pod, Knative allows outstanding requests to complete before terminating the pod. In this issue, a long-running request was being processed by a terminating pod. Knative miscalculated the request capacity of the revision: it accounted for the long-running request that was running on the terminating pod but did not account for the capacity of the terminating pod. To maintain acceptable request capacity (i.e., positive capacity), the Autoscaler kept an additional pod sitting idle while the long-running request ran to completion.

3.2.5 Request Routing Bugs. Roughly 20% of bugs in our survey involved issues in how Knative routes requests to the available function pods within a revision. In these bugs, Knative's routing logic consistently sent requests to revision pods that were at or near capacity even while other pods in the revision remained idle; thus, Knative's request load balancing was worse than a naive round-robin approach. These bugs can impact application performance and degrade resource utilization since the bugs prevent revisions from making effective use of their configured pods.

Examples. Issue #12593 describes a scenario in which all revision traffic is routed to a single pod. In this issue, a developer configured Knative to maintain a minimum 2 function pods per revision, with each pod configured to handle a single request at a time. A bug in the Knative Activator resulted in requests only being sent to one of the pods, where they would then queue up. This behavior drastically limited the utilization of the revision's resources and degraded average request latency.

Similarly, issue #11532 describes a scenario where two Activator components were deployed for high availability, and function pods were similarly configured to process one request at a time.

When two long-running requests were made to the revision, the first request began processing and the Autoscaler correctly provisioned a new pod to handle the second request. Yet, the second request was routed to the already busy first instance, leaving the newly scaled instance unutilized.

3.2.6 Version Semantics Bugs. A small, but not insignificant, percentage (7%) of bugs in our survey are related to Knative failing to correctly handle multiple revisions of the same function. As discussed in Section 2, Knative aims to reduce application downtime by automatically managing function revisions. Namely, the runtime can automatically "roll-back" a buggy revision to a previous revision. In a version semantics bug, the Knative runtime mismanages a function due to an incorrect assessment of the function's revision history, such as which revision was the most recent to have been marked as healthy. These bugs are especially frustrating for users, as a given version may contain critical bug fixes or business logic changes for a serverless application.

Example. Issue #12538 describes a scenario in which Knative and Kubernetes controllers interfere (Section 3.2.2) in a way that caused a healthy revision to be replaced by a previous version. In this case, a node preemption event set the Failed status condition for a revision. In response, Knative's configuration controller erroneously replaced the revision with an older version of the function, since it incorrectly determined that this older revision was the highest version known to be healthy. At the same time, Kubernetes' deployment controller replaced the preempted pod with a new one, thereby creating a new instance of the original revision. However, Knative's configuration controller had already updated the function's route configuration such that all traffic went to the older revision. Eventually, Knative's Autoscaler scaled the new revision to zero, as it received no traffic. In the end, the same preemption event caused both Knative and

Kubernetes controllers to interfere in a manner that resulted in (1) wasted resources, when the system had pods deployed for both revisions, and (2) the system converging to a stable but incorrect state in which an old revision was deployed instead of the most recent one. Knative developers fixed the issue by changing the Knative configuration controller so that it would not consider older revisions over newer ones in the case of a preemption. This fix still leaves the potential for similar bugs from interference between the Knative configuration and Kubernetes deployment controllers.

3.2.7 Invocation History Bugs. Finally, a few bugs in our survey (roughly 5%) are bugs that only occur when a function has a particular invocation history. In an invocation history bug, the runtime handles a function's failure differently if the runtime has executed the function before. In particular, the invocation history bugs in our survey involve the runtime behaving differently if the function has previously succeeded compared to if the function has never been executed. These bugs result in suboptimal resource utilization.

Example. Issue #6504 describes a scenario in which Knative leaves a failed revision in a crash loop indefinitely, provided that the function had previously been completed in the past.

If the function was not executed, then the runtime instead terminates the revision. This bug has not yet been fixed as of the writing of this paper.[3] The issue's discussion suggests that Knative developers are struggling to resolve invocation history bugs because Knative's design poorly handles pod issues that arise after the first pod invocation.

3.3 Takeaways

In addition to our bug taxonomy, we identify three takeaways from our survey. First, Knative's runtime semantics are difficult to specify; the first few developer comments in many issues illuminate a difficulty in even determining Knative's correct behavior. A formal semantics for Knative's complex runtime behavior would help Knative developers with this task. Second, many of Knative's bugs involve complex interactions across numerous system components interacting with a vast, and sometimes buggy, space of configurations. Manually testing this space is cumbersome and likely insufficient. Finally, even though Knative configuration bugs are *not* misconfigurations, Knative developers often "fix" them by changing default configurations. An automated and/or more principled approach to handling configuration issues is warranted. In Section 5, we discuss some directions for future work inspired by these takeaways.

4 RELATED WORK

Recent work has empirically studied the challenges developers face when building serverless applications. In a survey of Stack Overflow questions, Wen et al. [2021] provide a comprehensive taxonomy of developer challenges related to serverless computing. Their survey is not specific to any particular serverless platform, and is focused on the developer perspective; the majority of surveyed issues relate to application implementation rather than bugs in the runtime behavior of serverless platforms themselves, which is the focus of our work.

[3]And was opened in January 2020!

Recent work has also explored the design considerations and performance characteristics of serverless platforms including Knative. Kaviani et al. [2019] provide a comprehensive overview of Knative and compare its design with several popular serverless platforms. Their comparison seeks to explore what a common API layer for serverless might look like, noting that one has yet to emerge. Li et al. [2019] evaluate the performance of Knative relative to other open-source serverless platforms based on Kubernetes. The authors identify the design considerations with the greatest impact on baseline performance, namely the interface between the platform ingress and the function pods, and highlights insufficiencies in the auto-scaling approaches existing platforms currently employ. Mohanty et al. [2018] conduct a similar evaluation of open-source serverless platforms, focusing on request latency and platform availability under load.

5 DISCUSSION AND FUTURE WORK

Our bug survey highlights the challenges in implementing a performant and reliable serverless runtime on top of Kubernetes, the industry standard platform for deploying containerized applications.

With the line between serverless functions and containerized applications increasingly blurring [Datadog 2022], we believe that further research at the interface between serverless platforms and container orchestration platforms can contribute to the maturation of serverless programming at large. In this section, we suggest several directions for future work suggested by our survey.

5.1 Clarifying Runtime Semantics

Our survey reveals that in many places, correct runtime behavior remains underdefined. Many of the issues we reviewed describe peculiar or suboptimal behaviors, especially around the platform's request routing decisions (Section 3.2.5) and its handling of inactive function resources (Section 3.2.4). In these cases, the "bugginess" of the behavior was argued intuitively, and could not be discussed in terms of a well-understood specification being violated. An informal, prose specification of Knative does exist in the form of the Knative Runtime Contract [Knative Authors 2022], although it discusses behavior at too high a level to adequately describe many of the issues we reviewed in our survey. A comprehensive specification of Knative's runtime behavior has yet to be defined. Existing works [Burckhardt et al. 2021; Jangda et al. 2019] have aimed at giving a formal semantics to the serverless programming model, and could serve as inspiration for an eventual formal specification.

The Cloud Native Computing Foundation has articulated a general definition of what a serverless platform should consist of [CNCF 2019], but to our knowledge, no one has attempted to precisely define how one should behave. The discussions on the issues in our survey dataset may serve as a starting point for codifying a specification for a serverless platform's runtime behavior.

5.2 Automated Testing

We observe that Knative bugs are often caused by complex interactions across multiple system components (Section 3.2.2) within a huge, and even buggy (Section 3.2.3) configuration space. Consequently, Knative developers would benefit from automated testing

to help navigate a large and complex search space, similar to the approach taken by Sieve [Sun et al. 2022] to test third-party Kubernetes controllers. Such testing would help Knative developers limit regression bugs, and would also help mitigate the inherent incompleteness in our study. Moreover, Knative's architecture is conducive to automated testing since it is easy to instrument: the state-centric interface between Knative components and the overall state of the system readily supports automated testing strategies. A good starting point would be to find out to what extent existing automated testing tools such as Sieve [Sun et al. 2022] can apply to Knative, and explore which types of bugs Sieve might find in Knative, or how Sieve's approach would need to be altered to detect the bugs we study in this survey.

Many bugs also involved a particular ordering of status condition reads and updates (Section 3.2.1). Model checking could be used to evaluate controller coordination among the possible interleavings of these events, bringing the cost of thoroughly testing Knative's condition-based readiness model down to something more tractable. Such an approach could also be used to identify optimization opportunities, such as where Knative's readiness model can be relaxed safely to improve performance.

5.3 Improving Knative Configuration

While Knative configuration bugs are *not* misconfigurations, Knative developers nevertheless often "fix" these bugs by modifying default configuration parameters such that the issues impact fewer users. This observation suggests two avenues for future work to improve the correctness and efficiency of Knative.

First, existing automated misconfiguration detection and resolution tools, such as CTest [Sun et al. 2020] and PCheck [Xu et al. 2016], could find configuration parameters that lead to bugs. While these systems would not fix the bugs, adopting them in the development process would help developers to automate the process of identifying sensible default configurations. There is a research challenge in scaling such tools to Knative: Knative configuration issues are often caused by a combination of configuration values rather than by a single configuration (Section 3.2.3), so misconfiguration tools would need to explore a combinatorial search space of potentially buggy configurations.

Second, applying formal methods for verifying Knative configurations, similar to Rehearsal [Shambaugh et al. 2016], could mitigate Knative's configuration challenges by preventing serverless users from specifying configurations that Knative cannot support, although this approach would not suffice to address the *root causes* of these buggy configurations.

6 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful comments and suggestions in improving this paper. This material is based upon work supported by the National Science Foundation CSGrad4US Fellowship Program under Grant No. (2240204). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable Functions: Semantics for Stateful Serverless. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 133 (oct 2021), 27 pages. https://doi.org/10.1145/3485510

CNCF. 2019. *CNCF WG-Serverless Whitepaper v1.0.* Technical Report. Cloud Native Computing Foundation.

Datadog. 2022. *The State of Serverless.* https://www.datadoghq.com/state-of-serverless/

Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 149 (oct 2019), 26 pages. https://doi.org/10.1145/3360575

Nima Kaviani, Dmitriy Kalinin, and Michael Maximilien. 2019. Towards Serverless as Commodity: A Case of Knative. In *Proceedings of the 5th International Workshop on Serverless Computing* (Davis, CA, USA) *(WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3366623.3368135

The Knative Authors. 2022. *Knative Runtime Contract.* https://github.com/knative/specs/blob/main/specs/serving/runtime-contract.md

The Knative Authors. 2023. Knative. https://knative.dev/docs/.

The Kubernetes Authors. 2023a. Kubernetes. https://kubernetes.io/.

The Kubernetes Authors. 2023b. *Pod Lifecycle.* https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-conditions

Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. 2019. Understanding Open Source Serverless Platforms: Design Considerations and Performance. In *Proceedings of the 5th International Workshop on Serverless Computing* (Davis, CA, USA) *(WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/3366623.3368139

Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/mohan

Sunil Kumar Mohanty, Gopika Premsankar, and Mario di Francesco. 2018. An Evaluation of Open Source Serverless Computing Frameworks. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 115–120. https://doi.org/10.1109/CloudCom2018.2018.00033

Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A Configuration Verification Tool for Puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 416–430. https://doi.org/10.1145/2908080.2908083

Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 42, 17 pages.

Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing For Cluster Management Controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 143–159. https://www.usenix.org/conference/osdi22/presentation/sun

Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. An Empirical Study on Challenges of Application Development in Serverless Computing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 416–428. https://doi.org/10.1145/3468264.3468558

Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 619–634. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/xu