Reengineering Theodolite with the Java Operator SDK

Luca Alexander Mertens

Bachelor's Thesis
September 29, 2022

Software Engineering Group
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Wilhelm Hasselbring
Sören Henning, M.Sc.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den 30.09.2022

Luca Mertens

Abstract

Scalability is one of the most important quality characteristics of applications deployed in a distributed context. It is not trivial to predict how well a complicated system will scale under increasing load. Theodolite allows to efficiently benchmark the scalability of applications deployed on the container orchestration platform Kubernetes.
Internally, Theodolite extends the functionality of Kubernetes by implementing the Kubernetes operator pattern. Since Theodolite started adopting operators, the pattern has matured, leading to the emergence of several best practices and tools. One of these tools, the Java Operator SDK, aims to simplify the process of writing Kubernetes operators in Java or Kotlin.

In this thesis, we apply the reengineering process to Theodolite's operator: First, we analyze the state of the current operator and elicit its shortcomings. Second, we propose two architectures, one stateless and one stateful, for Theodolite's operator and discuss their advantages and disadvantages. Third, we implement the stateful architectures using the Java Operator SDK. We evaluate our implementation by comparing the current operator to the reengineered operator in terms of performance and memory usage. We find that the reengineered operator brings both a performance improvement and a reduction in communication overhead with the Kubernetes API. We also measure a slight increase in memory usage in the reengineered operator.

Contents

Chapter 1

Introduction

1.1 Motivation

Theodolite [Henning and Hasselbring 2021] is a scalability benchmarking tool for cloud native applications that are deployed on the container orchestration framework Kubernetes. Theodolite implements its core functionality in the form of a Kubernetes operator [Dobies and Wood 2020]. The Kubernetes operator pattern is a method of extending the functionality of Kubernetes, by automating application-specific tasks (e.g., installation of related components). For example, Theodolite utilizes the operator pattern to offer a declarative user interface (in the form of custom Kubernetes resources).

Since Theodolite started adopting Kubernetes operators, the pattern has become increasingly popular and mature, leading to the emergence of several best practices and tools. One of these tools, the Java Operator SDK [Red Hat, Container Solutions 2022], aims to simplify the process of writing Kubernetes operators in Java or Kotlin.

Reengineering Theodolite's current operator with the Java Operator SDK could have a positive effect on the following quality attributes:

Usability and Observability Since the operator controls a large part of Theodolite's user interface, reengineering it with observability in mind could lead to a more intuitive user experience. V. Kistowski et al. [2015] name usability as an important quality characteristic of benchmarking tools.

Simplicity and Maintainability Since a lot of general operator logic can be delegated to the SDK, the Operator will most likely become simpler and therefore easier to maintain. Furthermore, the Java Operator SDK enforces a certain structure on the operator, which could lead to a more consistent code base.

Thread-Safety and Stability The Java Operator SDK promises some thread-safety properties by default. Additionally, the reengineering process will involve creating an architectural description including lifecycle definitions for components, which can make edge cases more visible.

1.2 Goals

The following goals define the scope of this work.

G1: Present the Current State and Limitations of the Theodolite Operator

The current implementation of Theodolite and its shortcomings are analyzed and presented. We focus on shortcomings that can be solved by modifying Theodolite's operator, and address them in G2.

G2: Reengineer Theodolite's Operator Architecture with the Java Operator SDK

To realize the new operator, we follow the reengineering process, which we present in more detail in Section 2.1. In short, the process consists of the following steps:

1. **Reverse engineering**: An architectural overview over Theodolite's current operator is obtained from the source code. This step is already partly covered by G1.
2. **Restructuring**: An architectural description of the reengineered Operator is created and presented. Care must be taken to ensure that the architecture supports the usage of the Java Operator SDK.
3. **Forward engineering**: The devised architecture is implemented.

G3: Performance Evaluation

The performance of the reengineered operator is evaluated by comparing it to the current operator. For this, multiple scenarios are simulated. We measure the average CPU-usage, the average memory usage and the average load on the Kubernetes API.
The three goals are also used to guide the structure of this thesis, which is described in the following section.

1.3 Document Structure

Chapter 2 introduces foundations and technologies that are relevant for understanding the rest of this thesis. Chapter 3 gives an overview over the current operator and outlines its shortcomings. In Chapter 4, two possible architectures for the Theodolite operator are presented, along with a discussion of their advantages and disadvantages. We implement one of the architectures, and present the results in Chapter 5. In Chapter 6, we determine a suitable source of metrics and compare the performance of the reengineered operator to the current one. Related work is presented in Chapter 7. Finally, Chapter 8 concludes this thesis and presents possible future work related to Theodolite's operator.

Chapter 2

Foundations and Technologies

2.1 The Software Reengineering Process

Chikofsky and Cross [1990] define reengineering as the process of analyzing and modifying existing software in order to improve a quality attribute, often maintainability. They also propose a collection of methods for the reengineering process, where the methods are "transformations between or within abstraction levels" [Chikofsky and Cross 1990]. The most important methods are visualized in Figure 2.1 and described in the rest of this section.

2.1.1 Reverse Engineering

Reverse engineering describes the process of obtaining a representation of a system that is on a higher abstraction level than the current representation [Chikofsky and Cross 1990].

The authors name two subareas of reverse engineering: **Redocumentation**, which involves only observing the implementation to construct a documentation, and **redesign**, which additionally involves using external knowledge to create a more abstract design documentation.

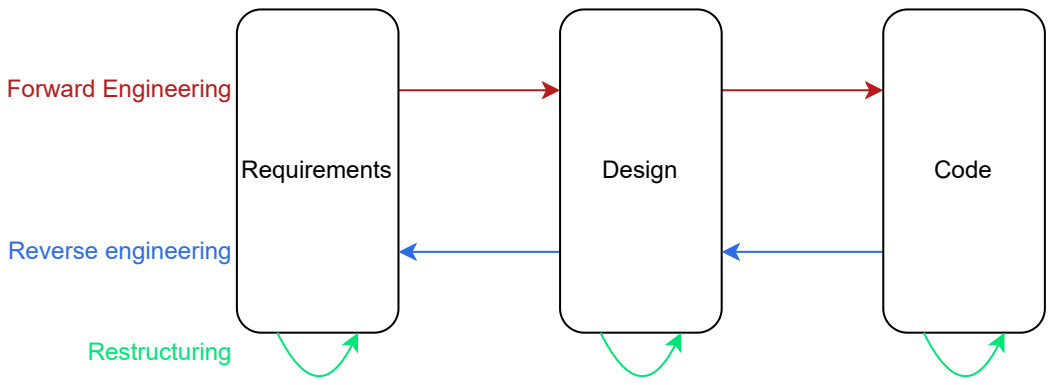


Figure 2.1. The reengineering process (adapted from Chikofsky and Cross [1990]).

2.1.2 Restructuring

Restructuring describes the process of modifying the system while staying on the same layer of abstraction. An example of this is the modification of an existing architecture description for a system.

2.1.3 Forward engineering

Forward engineering describes moving from a higher level of abstraction to a lower one. Typically, a description of a system's envisioned architecture is used to implement said system. In the Reengineering process, this architecture description emanates from the previous two steps.

2.2 The Container Orchestration Platform Kubernetes

The Kubernetes Authors [2022e] describe Kubernetes as an "open source platform for managing containerized workloads and services". Important features include automatic load balancing, if there are multiple replicas of the same service, and self-healing, i.e., restarting / replacing failing containers.

Kubernetes offers a declarative interface: The user defines a desired state, which Kubernetes then tries to realize (based on the current state). As an example, if the user specifies a desired state of 3 replicas of an application, Kubernetes will start the 3 instances of the application that the correct amount is always present, recreating them on failure. To accomplish this realization of desired state, Kubernetes uses the controller pattern, which is described in the following subsection.

2.2.1 The Kubernetes Controller Pattern and Control Loops

Controllers [The Kubernetes Authors 2021] watch the desired and actual state of the cluster and apply the logic necessary to actualize the desired state. Typically, a controller *controls* a single Kubernetes resource type, but it's also possible for a controller to manage multiple resource types, including resources that are not part of the Kubernetes cluster. The controller pattern is both employed Kubernetes-internally (example: the controller for the `Job` resource), and -externally, in the form of user-written custom controllers.

Controllers are mostly structured with **control loops** (often also called **reconciliation loops**). According to Hausenblas and Schimanski [2019], a generic iteration of such a control loop consists of the following parts:

1. Read the current state of the cluster.
2. Trigger a change of state (e.g., launch a pod).
3. Update the displayed status of the changed resource.

We present different ways of implementing control loops in the following subsection.

2.2.2 Edge- and Level-Driven Control Loop Triggers and Logic

Hausenblas and Schimanski [2019] name two important factors in the design of control-loops for **Kubernetes-Controllers**: How control loops are triggered and what information their logic is based on. They describe them using terms usually found in systems programming:

Edge-driven trigger The control loop is executed whenever an event is received.

Level-driven trigger The control loop is executed periodically, e.g., every 2 seconds.

Edge-driven logic Any information about the cluster used inside the control-loop is deducted from the content of received events.

Level-driven logic Any information about the cluster used inside the control-loop is queried from the cluster.

2.2.3 The Kubernetes Operator Pattern

The term "Kubernetes operator" was initially coined by Red Hat [2022] and is conceptually similar to Kubernetes controllers. Kubernetes operators encapsulate one or more controllers with some kind of operational knowledge [Dobies and Wood 2020]. The main difference between controllers and operators is that controllers are typically written with a less specific use case in mind (e.g., managing a single type of Kubernetes resources), while operators are usually specific to their application and include domain knowledge.

For example, consider a replicated database that only guarantees eventual consistency. If the user would like to reduce the number of replicas, additional measures must be taken to ensure that the deleted replicas do not contain any records that have not yet been synchronized to at least one other replica. Since the lifecycle is not just limited to Kubernetes-scoped logic but also includes some application-specific knowledge, the controlling software for the database would be classified as an operator.

2.2.4 Kubernetes Informers

Informers are useful if a controller wants to efficiently stay *informed* about changes to a certain type of resource. They are commonly used in Kubernetes' internal controllers; Hausenblas and Schimanski [2019] even call them "one of the main architectural concepts in the Kubernetes API design". Informers are instantiated within the runtime-environment of the controller, so it is possible to utilize them in custom controllers as well.

The Kubernetes API offers so-called *watch* requests [The Kubernetes Authors 2022a]: When a client sends a watch request to the API, the API will keep the connection open and

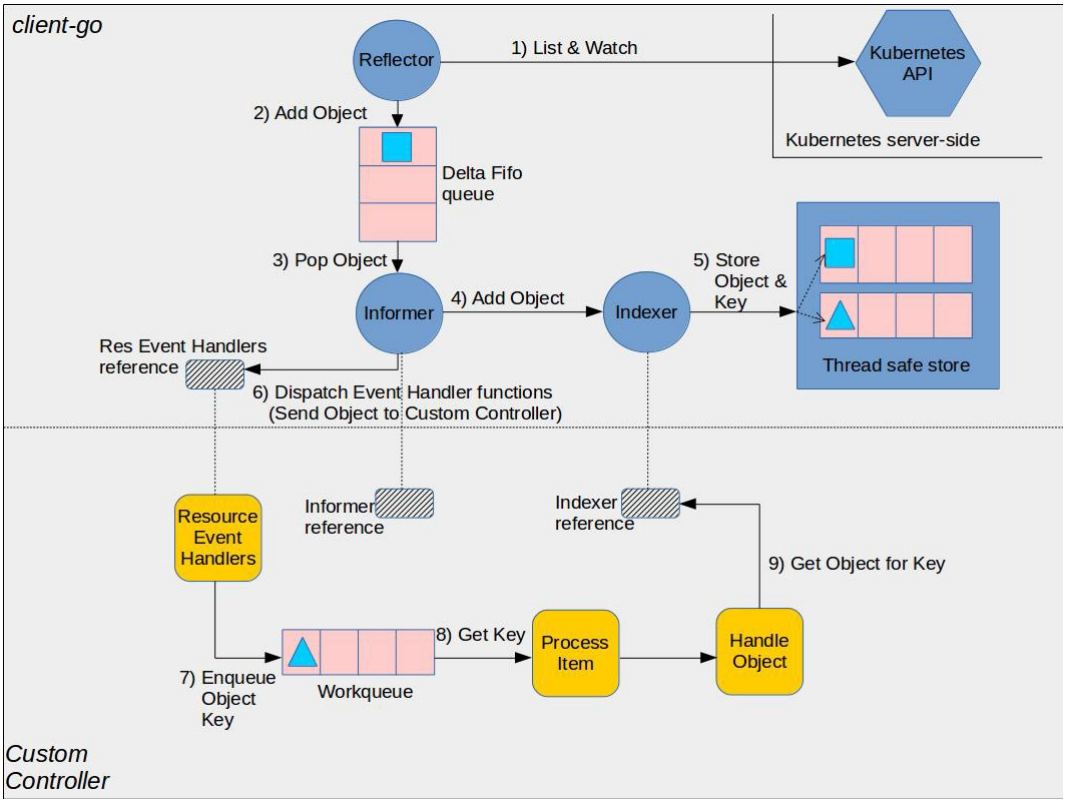


Figure 2.2. Informers watch the Kubernetes API (1), update their local state by processing the delta (2,3), cache the retrieved elements (4), update local indexes (5), and notify event handlers (6). Diagram by The Kubernetes Authors [2018].

send an update whenever a resource of the referenced type is created, updated or deleted. This corresponds to the Observer pattern.

Informers use watch requests to act as a local cache for a certain type of Kubernetes resource. They can be used to efficiently retrieve the current state of the custom resource they are watching, without querying the Kubernetes API. Furthermore, they can notify controllers whenever a change has occurred. Their functionality is visualized in Figure 2.2. A major advantage of informers is that they can be shared across the whole operator-runtime. If multiple controllers need to retrieve the current state of a custom resource, they can simply all query the shared informer, avoiding superfluous calls to the Kubernetes API.

The Java Operator SDK requires controllers to always retrieve the current state of the cluster for their control loop logic (for more information, see Figure 4.1). Therefore, a caching mechanism like informers is critical to the performance of operators written with

the SDK.

Informers offer another convenience feature: Indexing. If the user provides a function of type `indexer :: CustomResource -> [String]`, the informer will automatically generate a key-value map (the index) of type `String -> CustomResource`. This index is updated whenever the watched custom resource changes.

2.3 The Scalability Benchmarking Framework Theodolite

Theodolite [Henning et al. 2021] is a scalability benchmarking Framework for cloud-native applications. More specifically, it can be used to measure the horizontal and vertical scalability of a containerized software system in a Kubernetes cluster [Henning and Hasselbring 2022]. It implements the operator pattern to extend Kubernetes, as depicted in Figure 2.3.

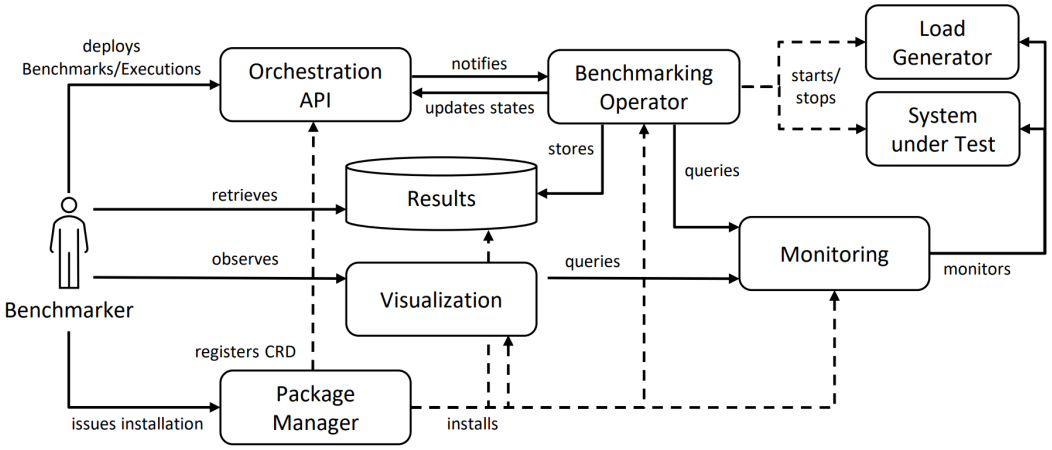


Figure 2.3. An overview over Theodolite's current architecture [Henning and Hasselbring 2022]. For this work, the interaction of the operator with other components is of special interest.

To run a benchmark with Theodolite, the user needs to provide a `Benchmark` and an `Execution`, as described in the following.

2.3.1 The `Benchmark` Custom Resource

`Benchmarks` only describe the application that will be benchmarked, but do not include any configuration for the actual benchmarking process.

Their most important properties are:

System Under Test (SUT) Information about how to deploy the application that is to be benchmarked.

Load Generator A Kubernetes deployment that generates a configurable amount of workload on the SUT.

Service Level Objectives (SLOs) A (testable) definition, defining whether the SUT can deliver its service under a given amount of load.

Benchmarks can be executed multiple times by deploying accompanying `Execution` resources.

2.3.2 The `Execution` Custom Resource

`Executions` describe a specific instance of a benchmark. The actual benchmarking process is only started when an execution has been deployed to the cluster. `Executions` include additional configuration, for example:

Resource dimensions A discrete numeric range describing different amounts of resources available for the SUT, e.g., replicas. They are depicted by the values on the x-Axis in Figure 2.4.

Load dimensions A discrete numeric range describing different amounts of load on the SUT, e.g., requests per second. They are depicted by the values on the y-Axis in Figure 2.4.

Search strategy A strategy to determine which SLO-experiment (resource amount, load amount) should be run next. See Figure 2.4 for examples.

Scalability metric Currently, Theodolite supports two different scalability metrics, as defined by Henning and Hasselbring [2022]:

- The *demand* metric, which, for a given load amount (*demand*), finds the minimum amount of resources necessary for the system to fulfill its SLO.
- The *capacity* metric, which, for a given amount of resources (*capacity*), finds the maximum load under which the system can still fulfill its SLO.

2.4 The Java Operator SDK

The Java Operator SDK [Red Hat, Container Solutions 2022] aims to simplify the development of efficient Kubernetes operators in Java. Internally, it uses the Kubernetes-API-Client Fabric8 to communicate with Kubernetes' REST-API. According to the project's website, the SDK's key features are:

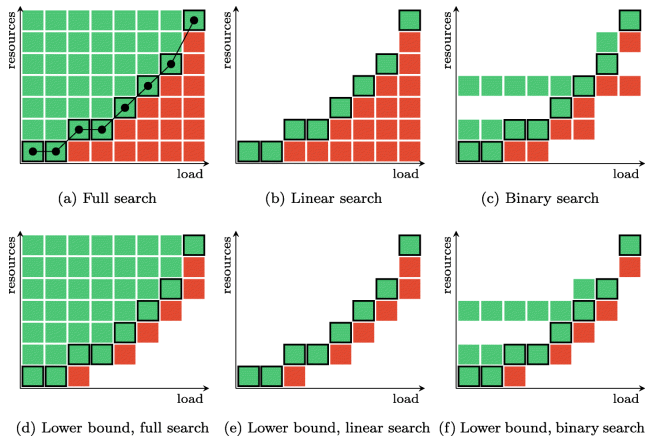


Figure 2.4. A visualization of different search strategies that can be used by Theodolite to more efficiently determine the scalability of the system under test. A red square means that the SLO was not fulfilled for the given resource and load amount. Diagram by Henning and Hasselbring [2022].

Concurrency control for event processing The SDK allows the user to define a `depends on` relation to create a dependency graph for Kubernetes resources. Using this, related events can be processed sequentially, and unrelated events can be processed concurrently.

Kubernetes internal and external events The SDK provides an extensible API to let the user define reconciliation triggers, so-called *event sources*. This allows the reconciler to not only react to Kubernetes-internal events (e.g., a resource being created or updated), but also to external events (e.g., a webhook provided by an application-component external to the cluster).

Simplify the usage of informers The SDK simplifies the act of efficiently interacting with the Kubernetes API: For custom resources managed by the operator, the SDK automatically creates shared informers (see Section 2.2.4) to cache the relevant resources. Additionally, it simplifies the creation of custom informers, e.g., by letting the user create `InformerEventSource` for watching different Kubernetes resources.

Automatically retrying failed operations Kubernetes implements optimistic locking for resources. This means that if the Kubernetes API receives multiple concurrent write instructions for a specific resource, it only processes the first one and rejects the others. The Java Operator SDK ensures that the rejected write attempts are retried automatically.

Smart event scheduling Kubernetes events are typically processed *level-driven* (see Section 2.2.2), meaning that the control loop reads the current state from the API, as opposed to processing event deltas. Therefore, the Java Operator SDK can apply some optimizations, such as only processing the last event for a type of resource.

Chapter 3

The Current Operator (Reverse Engineering)

As mentioned in Section 2.1, the first step in the reengineering process involves moving from one level of abstraction to a higher one. In our case, this means obtaining a design description of the current operator from the source code.

In the following chapter, we will focus on how the operator interacts with the Kubernetes API to manage its custom resources, how the benchmarking logic is structured and finally which shortcomings (with respect to common quality attributes) were found in the current implementation.

3.1 Custom Resource Management

As described in Section 2.3, the user needs to deploy two custom resources in order to start a benchmark (a `Benchmark` and an `Execution`). In the current version of Theodolite, these two different resources are controlled by multiple different concurrent control loops, each with their own purpose. The control loops are listed and visualized in Figure 3.1.

For example, `Executions` are controlled from three different sources. Each source is started in its own thread, and different styles of control loop triggering and control loop logic (as introduced in Section 2.2.2) are used.

3.2 Benchmarking Logic

3.2.1 Search Strategies

As described in Section 2.3, search strategies determine which SLO-experiments are run during an execution. They separate the benchmarking logic from details of how SLO-experiments are executed.

Search strategies are also the main part of the operator's locally maintained state: A chosen strategy is instantiated at the start of an execution and stays persistent for its entire duration. The next experiment is always determined based on previous experiment results and on the internal state of the strategy. The form of this internal state varies across different types of search strategies. For example, the naive `FullSearchStrategy` does not

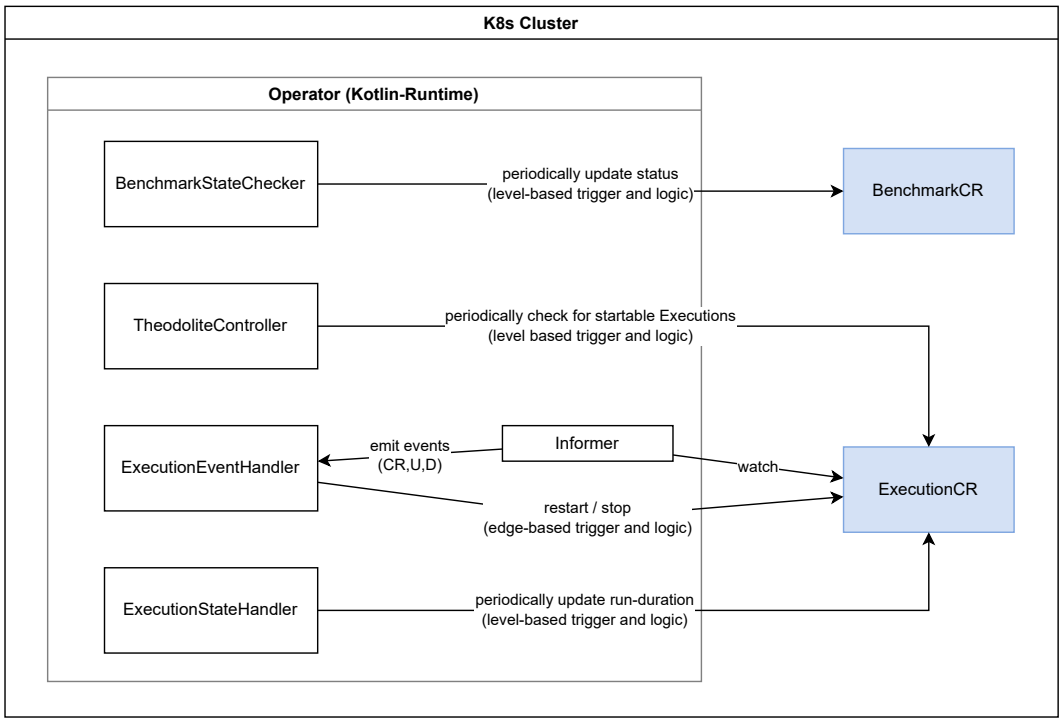


Figure 3.1. An overview of interactions between the operator and the Kubernetes custom resources.

need to maintain any extra state at all, while the `BinarySearchStrategy` implicitly maintains some context information on the call stack.

This property makes it hard (but not impossible, as shown in Section 8.2.1) to store the state of strategies in the Kubernetes API instead of in the operator runtime.

3.2.2 The Theodolite Executor

Once both a `Benchmark` and an `Execution` are deployed and in the `Ready` state, it is the operator's responsibility to execute the actual benchmarking logic. To do so, it starts a `TheodoliteExecutor` in a new thread. A typical run of the `TheodoliteExecutor` is shown in Figure 3.3.

Currently, the logic encapsulated by the `TheodoliteExecutor` is sequential and not suspendable, as visualized in Figure 3.2. This means that the operator can only start or stop a running execution as a whole, but has no control over individual SLO-experiments. This yields the advantage that the logic required to actually run the execution can be written in a sequential and imperative style (i.e., as a collection of Kotlin classes), which favors simplicity. However, this also entails some downsides, as described in the next section.

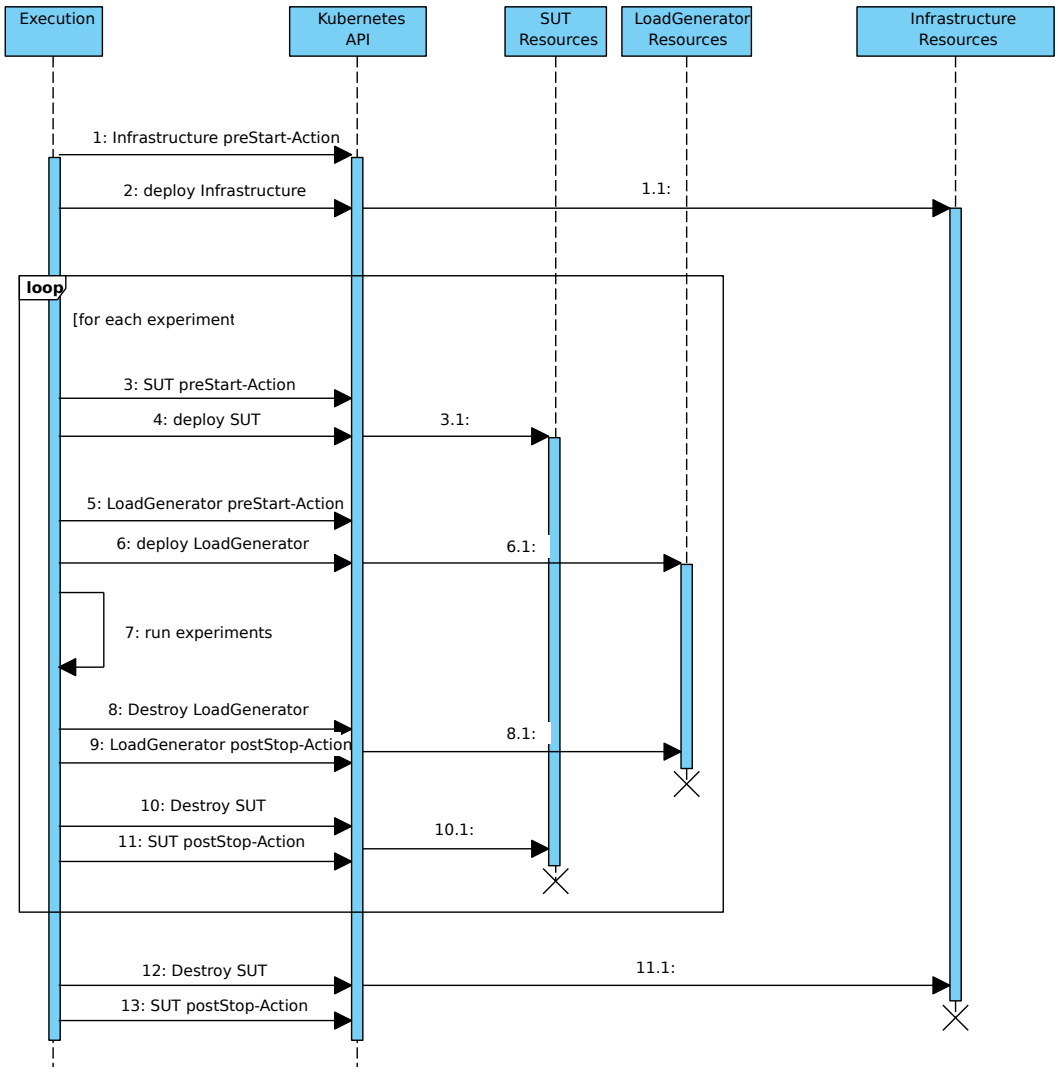


Figure 3.2. A typical execution of the benchmarking logic. Return messages omitted for clarity.

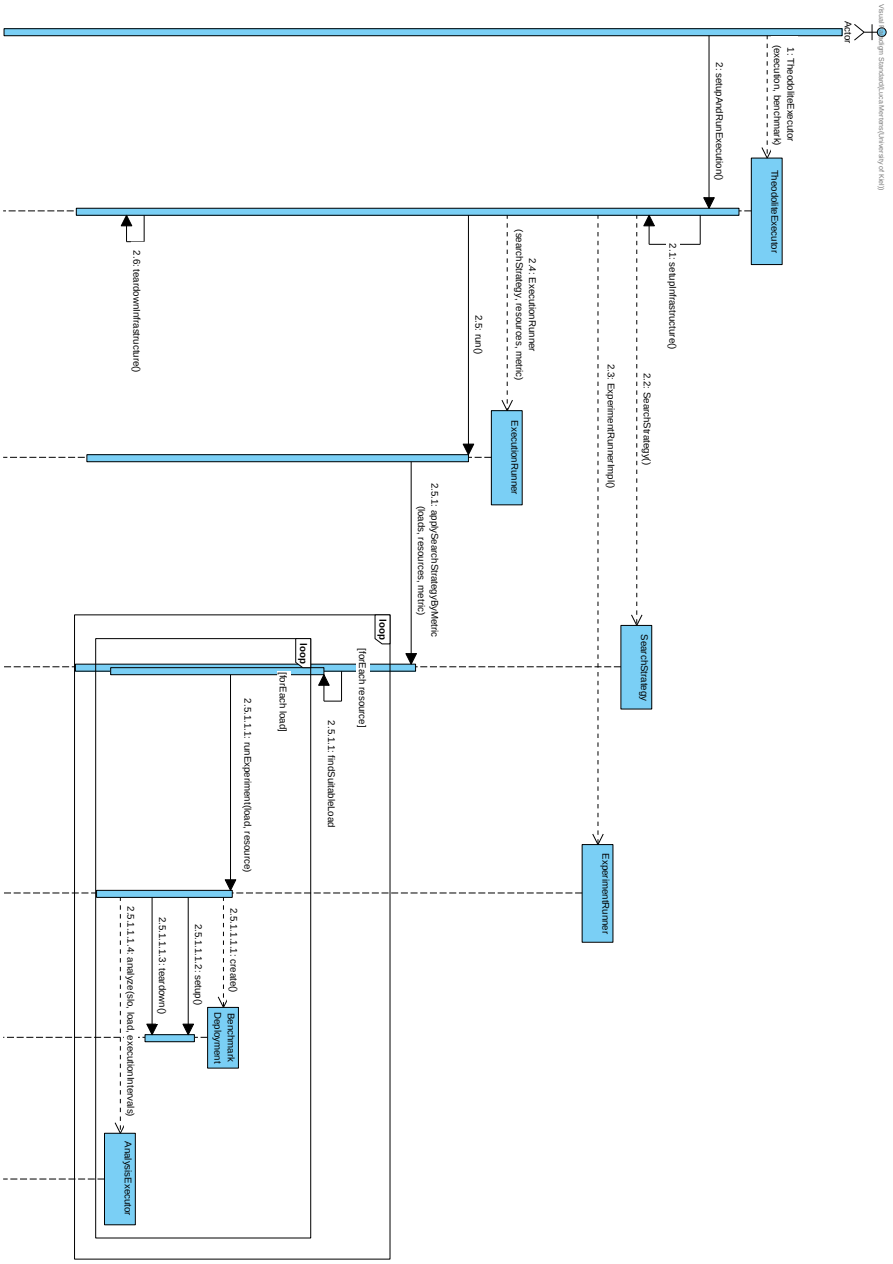


Figure 3.3. A typical run of the `TheodoliteExecutor`. Irrelevant classes have been omitted, and most methods have been simplified.

3.3 Summary of the Elicited Shortcomings

Efficiency Due to the heavy usage of polling, control loop iterations are frequent, even though relevant changes to custom resources typically occur relatively sparsely. This can impose an unnecessarily high load on the Kubernetes API, since it is queried in every iteration.

Consistency The operator relies on event details to detect if an `Execution` has changed in the middle of a running benchmark. This can leave the operator in an inconsistent state, if events are lost or arrive in the wrong order.

Additionally, the control loops responsible for `Executions` each run in a separate thread. Therefore, write-requests to the same custom resource can happen concurrently. Since the Kubernetes API implements a pessimistic locking mechanism, only the first write will succeed. If the rejected write was issued by a control loop with level-driven logic, the write will simply be retried in the next iteration. However, if the rejected write was issued by the control loop with edge-driven logic, the write will not be retried, leaving the `Execution` in an inconsistent state.

Fault tolerance Executions can be relatively long: Though Henning and Hasselbring [2022] recommend short durations of SLO-experiments ($\leqslant 5min$) and few experiment repetitions ($\leqslant 5$), even short experiments can easily sum up to an execution duration of several hours. It is usually desirable not to maintain any local state in the operator for such long periods of time, since operators can be (spuriously) stopped and restarted at any time (e.g., when being moved to another Kubernetes node). After such an event, the execution-run would need to be restarted from the beginning. This situation seems to occur rarely in practice, but still leads to a bad user experience if it does happen.

Observability The current state of a running execution only exists in the `Executor` and is only partially exposed to the user through events.

Understandability Since the status of `Executions` is updated from different sources, it is hard to reason about the state of the custom resource and to ensure that the status is always consistent.

Chapter 4

Architecture of the Reengineered Operator

4.1 Architectural Decisions Predetermined by the Java Operator SDK

The Java Operator SDK aims to simplify the process of writing Kubernetes Operators [Red Hat, Container Solutions 2022]. Therefore, the SDK imposes a certain structure on the operator, as illustrated by Figure 4.1.

4.2 Reconciliation Triggers and Logic

The different kinds of reconciliation triggers and reconciliation logic were presented in Section 2.2.2. Hausenblas and Schimanski [2019] advise using edge-driven triggers and level-driven logic, combined with an additional maximal interval between reconciliations (automatically reconcile if no event was received for a certain amount of time). The reason for not using edge-driven logic is that, since we are in a distributed context, events are not guaranteed to be delivered, which results in an incorrect representation of the cluster's state in the controller. The usage of edge-driven triggers reduces the load on the Kubernetes-API, because frequent polling can be avoided. However, if an event is lost, the controller will not react to the change of state in the cluster until another event occurs or until the infrequent periodic reconciliation is triggered. During this time, the resources managed by the controller remain in an outdated state.

Red Hat, Container Solutions [2022], suggest a similar approach. Hence, the Java Operator SDK does not offer access to the content of received events, preventing edge-driven logic altogether.

If implemented naively, level-based logic can impose a high load on the Kubernetes API, because the current state is requested in each reconciliation-iteration. To counteract this issue, the Java Operator SDK offers support for informers (see Section 2.2.4).

Using level-driven logic also requires that the reconciliation loop is **idempotent**, meaning that multiple reconciliation-iterations should not yield different result, if the cluster state or internal state has not changed between the iterations.

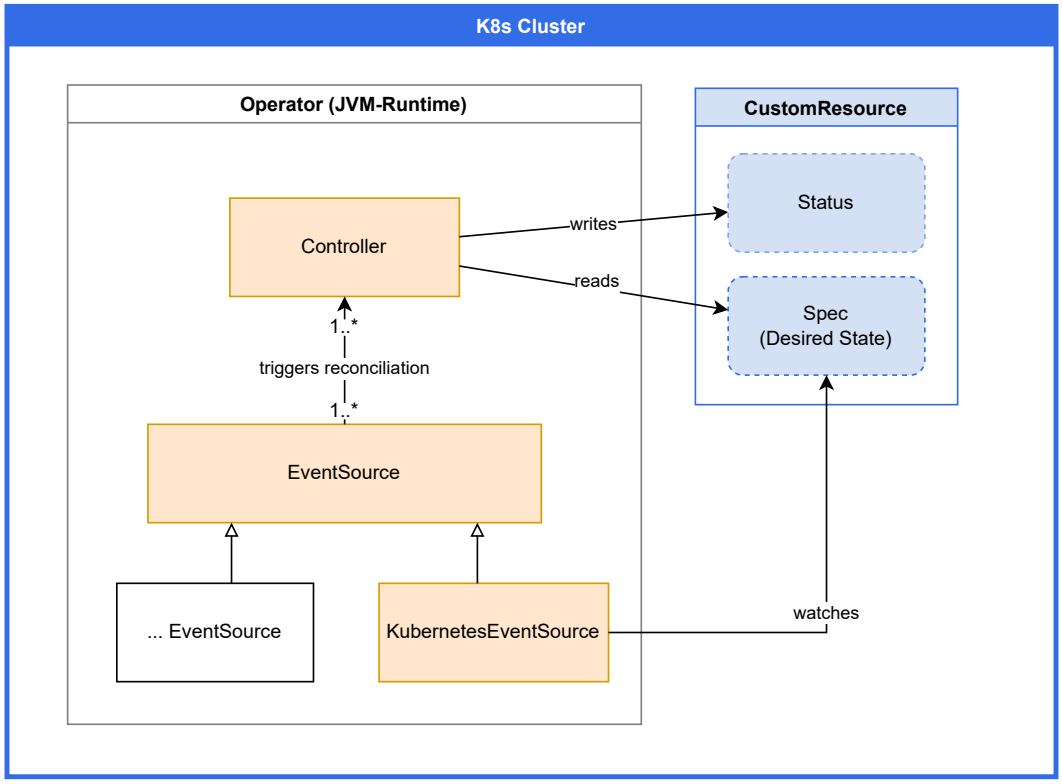


Figure 4.1. The general structure of an operator written with the Java Operator SDK.

4.3 Considered Approaches

When developing a concurrent algorithm, it is often helpful to start by decomposing the problem into smaller tasks that can potentially be executed in parallel. Even though parallelization is not feasible for Theodolite Benchmarks, the notion of task decomposition is still useful in the context of Benchmark Executions. We previously presented the execution of a benchmark as a sequence of steps (see Figure 3.2). The smallest possible unit that executions can be decomposed into are so-called experiments. An experiment tests whether the system can fulfill its SLO for a given load and resource amount.

In the following, we present two approaches, which mainly differ in the way executions are decomposed into tasks.

4.4 Approach 1: Stateful Operator

In this approach, the logic required to run a benchmark is not decomposed into smaller units. Rather, the benchmarking logic is executed as a single unit by a separate component within the operator (named `LocalRunner` here).

The rationale for extracting the benchmarking logic into a separate component is the following: The Java Operator SDK recommends that, for controllers to stay reactive, reconciliations should be kept short. The reason for this is that reconciliation is always synchronously finished before any other lifecycle components are executed[1]. In our concrete situation, this means the following: If we were to synchronously execute the long-running benchmarking run in a reconciliation loop, the task would not be cancelable from the outside. Therefore, benchmark runs need to be started asynchronously from the reconciliation loop, and monitored in subsequent iterations. As a consequence, this approach (and all its sub-approaches) contain a component that asynchronously executes the benchmark run (`LocalRunner`).

We present pro- and contra-arguments, as well as 3 concrete sub-approaches, which mainly differ in the way information is distributed across the cluster.

Main Arguments For This Approach

Easier to understand The benchmarking logic is sequential by nature and is therefore simpler to implement and understand using sequential logic.

Low implementation risk It is unlikely that a yet unrecognized factor prevents this approach from being feasible.

Reuse of existing code Since the current implementation of Theodolite already implements sequential benchmarking logic, a lot of it could be reused with little adaption.

Main Arguments Against This Approach

Not observable by default Additional effort must be made to make the current status (e.g., completed experiments) of the local runner available to the user.

Not fault-tolerant by default Ideally, Kubernetes operators should be stateless [Red Hat, Container Solutions 2022]. Since the operator holds local state while the execution is running (encapsulated by the `LocalRunner`), the execution cannot be resumed after an operator failure.

Concurrency management Additional care must be taken to ensure that there is only one local runner per cluster (i.e., that only one execution is running at a time).

[1] A feature proposal to change this behavior was opened by the author of this thesis on `GitHub`, but it was rejected because stopping a running reconciliation would entail a substantial amount of complexity.

4.4.1 Approach 1.1: Local Runner, Update Status Directly

The `ExecutionReconciler` asynchronously starts a `LocalRunner`, which in turn directly communicates with the Kubernetes API to update the status of the `Execution` custom resource. The modification of the custom resource's status then triggers the reconciler again. The reconciler will ensure that the resources' status is consistent and manage the lifecycle of the `LocalRunner`. A major advantage of this approach is its simplicity. However, Red Hat, Container Solutions [2022] discourage updating custom resources managed by the Java Operator SDK from other places than the reconciler, because this prevents the SDK from efficiently maintaining an internal representation of the resource. Furthermore, the direct change of status will trigger a reconciliation, which will be unnecessary in most cases.

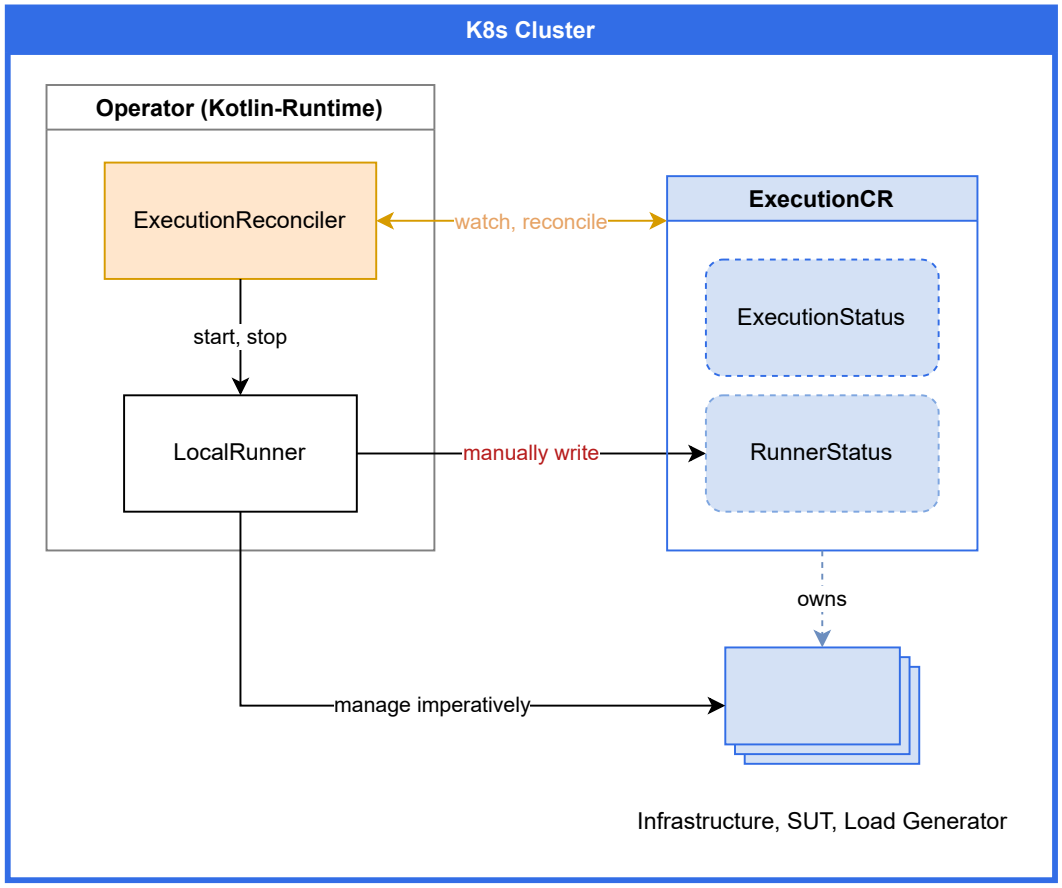


Figure 4.2. Approach 1.1: The local runner directly updates the status of its `Execution`.

4.4.2 Approach 1.2: Local Runner, Update Status Indirectly

To avoid directly updating the `Execution`'s status from the `LocalRunner`, the `ExecutionReconciler` is notified whenever the status of the `LocalRunner` changes. It then updates the `Execution` itself. The status is only updated from the `reconcile()` method, which simplifies debugging.

Usually, the communication between the `ExecutionReconciler` component and the `LocalRunner` component could be implemented using edge-driven logic (i.e., traditional event-driven communication). However, since the `reconcile` method's logic needs to be level-driven (see Section 4.2), the `LocalRunner` must provide a shared state object that can be read by the `ExecutionReconciler`.

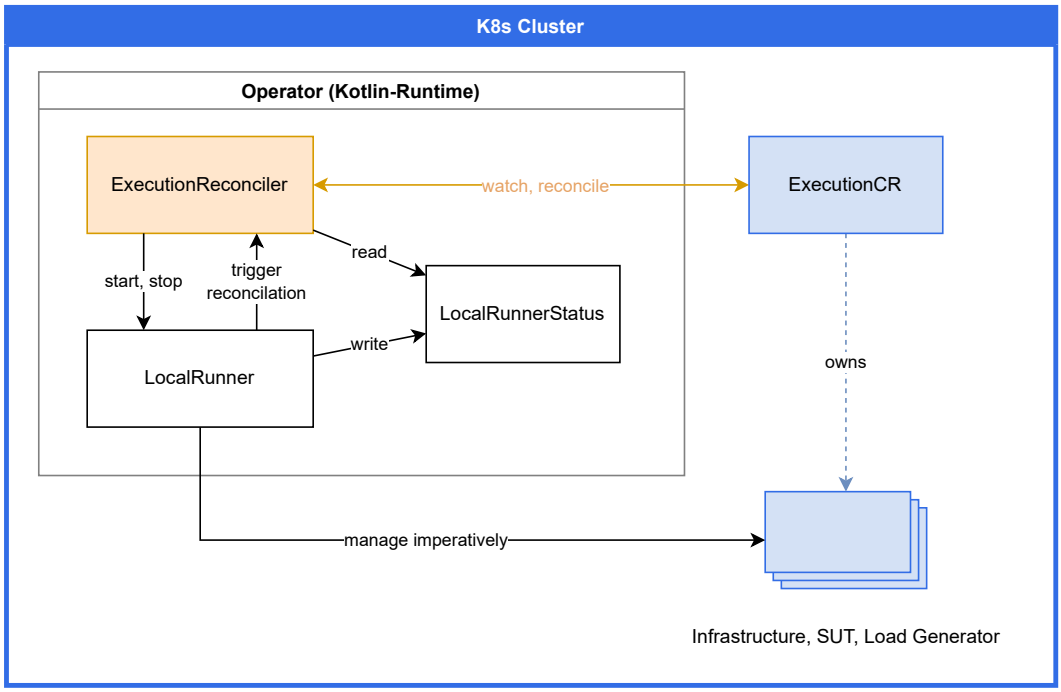


Figure 4.3. Approach 1.2: The `Execution`'s status is only updated from its reconciler.

4.4.3 Approach 1.3: Separate Custom Resource for Runner

The `LocalRunner` is wrapped in a Kubernetes custom resource. This simplifies repeating failed execution runs, because the `ExecutionRun` resource can simply be deleted and recreated, without having to delete the `Execution` resource. Since it directly owns all resources related to the experiment, these resources will automatically be garbage-collected by Kubernetes in that case.

The relation between `Executions` and `ExecutionRuns` is comparable to the relation between `Deployments` and `ReplicaSets`:

`ReplicaSets` ensure that a specified number of pods with a certain label is available; creating pods, if there are not enough, or deleting excess pods [The Kubernetes Authors 2022c]. If the user would like to update the pod specification, they would have to manually scale down the `ReplicaSet` to 0, update its pod template, and then scale it back up. The same can be achieved on a higher level of abstraction, by using `Deployments`. A `Deployment` provides additional features such as rolling upgrades and rollback, by fully managing a `RelicaSet`.

The same pattern is applied in this approach. The `ExecutionRun` is a low-level resource that only tries to run the execution once. `Executions` serve as a configuration unit for `ExecutionRuns` and provide a higher level of abstraction, by managing the lifecycle of `ExecutionRuns` to add features such as automatic retries.

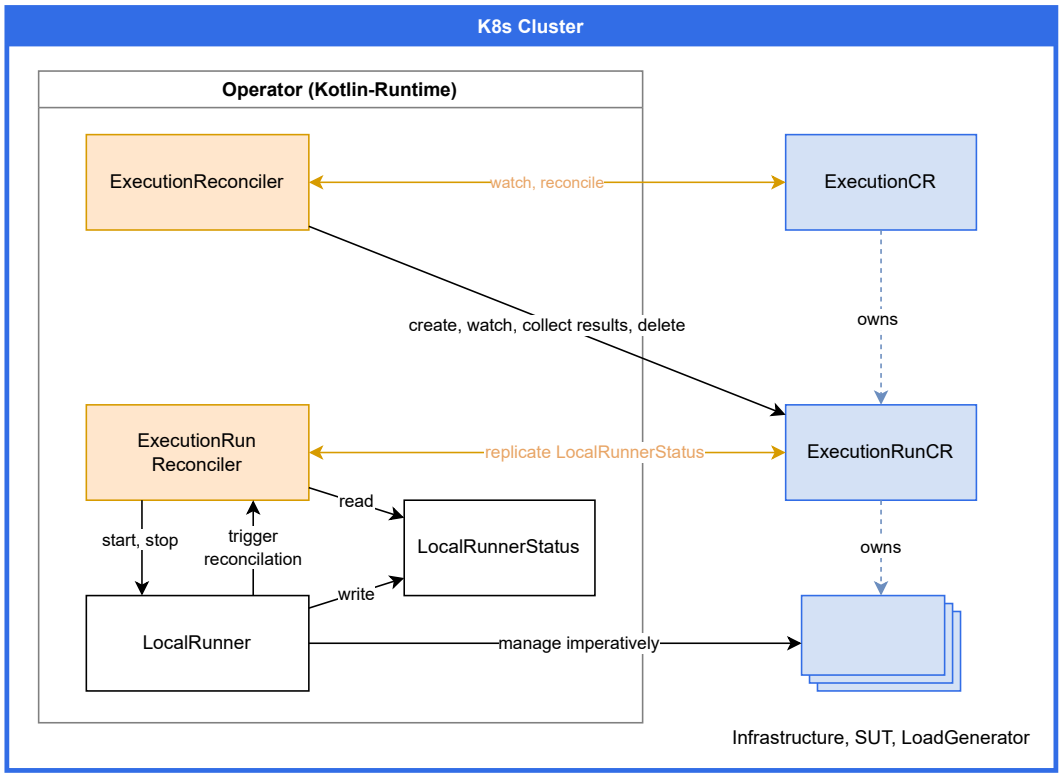


Figure 4.4. Approach 1.3: The `LocalRunner` is represented by a CR.

4.5 Approach 2: Stateless Operator

For this approach, the execution logic is decomposed into its individual SLO experiments. For each experiment, the execution reconciler creates a separate Kubernetes custom resource (`ExperimentRun`). The only job of the `ExperimentRunReconciler` is to execute the experiment for a given load and resource amount. This can be done synchronously in the reconciliation loop. This approach also enables the operator to be essentially stateless, meaning that necessary state, for example the result of previous experiments, is saved in the `Execution` custom resource.

The approach is visualized in Figure 4.5.

Advantages Related to the Operator Being Stateless

More fault-tolerant by default Since the operator itself is stateless, it can be arbitrarily stopped and restarted, without completely halting the benchmarking logic (i.e., the `Execution` can be resumed after a temporary operator failure). Furthermore, individual experiments can more easily be repeated in case of a temporary fault (e.g., a network partition). This functionality would have to be manually added to an operator that implements approach 1, as described in Section 8.2.1.
See Section 3.3 for more details about fault tolerance in the Theodolite operator.

Horizontally scalable Since the operator is stateless, it is theoretically possible to deploy multiple instances of it. The Java Operator SDK allows multiple replicas of an operator, but only one of those replicas (the *leader*) actually processes events and reconciles resources[2]. The other replicas are only held in stand-by, and take over in case the leader fails. The purpose of this functionality is to provide a faster recovery from faults.

More observable by default Since the operator's entire state is stored in the Kubernetes resource, it is automatically visible to the user.

Advantages Related to Delegating Functionality to Kubernetes and the JOSDK

Delegating concurrency management By using the Kubernetes object count quota feature[3], we can guarantee that only one `Experiment` is deployed to the cluster at all times.

Additionally, the Java Operator SDK ensures at most one concurrent reconciliation per resource. If we execute the experiment logic synchronously in the `reconcile()` method, we can guarantee that only one experiment can be executed at a time, without implementing any logic ourselves.

Automatic garbage collection As stated in Section 2.3, Theodolite allows users to define different types of resources, which have different lifespans: Infrastructure resources are

[2] `https://javaoperatorsdk.io/docs/features#leader-election`

[3] `https://kubernetes.io/docs/concepts/policy/resource-quotas#object-count-quota`

persistent during the entire execution, while the SUT and load generator are deployed and destroyed as part of each experiment.
Representing each experiment as a Kubernetes resource allows for a more fine-grained assignments of Kubernetes ownership references possible (as visualized in Figure 4.5). This enables automatic garbage collection: If an experiment is deleted, its owned resources are automatically deleted by Kubernetes as well.

Main Arguments Against This Approach

Complete reimplementation Since this approach represents a completely different paradigm (declarative logic instead of sequential logic), it would entail a complete reimplementation of most core functionality of Theodolite. Combined with the introduction of the Java Operator SDK, this will most likely exceed the scope of a bachelor's thesis.

Inelegant / harder-to-understand workarounds necessary The logic for determining which experiment should be run next is inherently stateful, as discussed in Section 3.2.1. When storing the state of a search strategy in the `.status` field of the `Execution`, it would have to be reconstructed in each reconciliation. This would mean that the strategy has to be re-executed on the basis of previous results, until it demands an experiment that has not yet been executed.

Increased complexity The approach involves more moving parts (see Figure 4.5) and an increased communication effort, due to the introduction of a custom resource for each experiment.

Potential coherency problems Additional care must be taken to detect whether the execution resources' specification has been modified during an execution run. Otherwise, it is possible that some experiments are executed on the basis of the old execution specification and some are executed on the basis of the new specification.

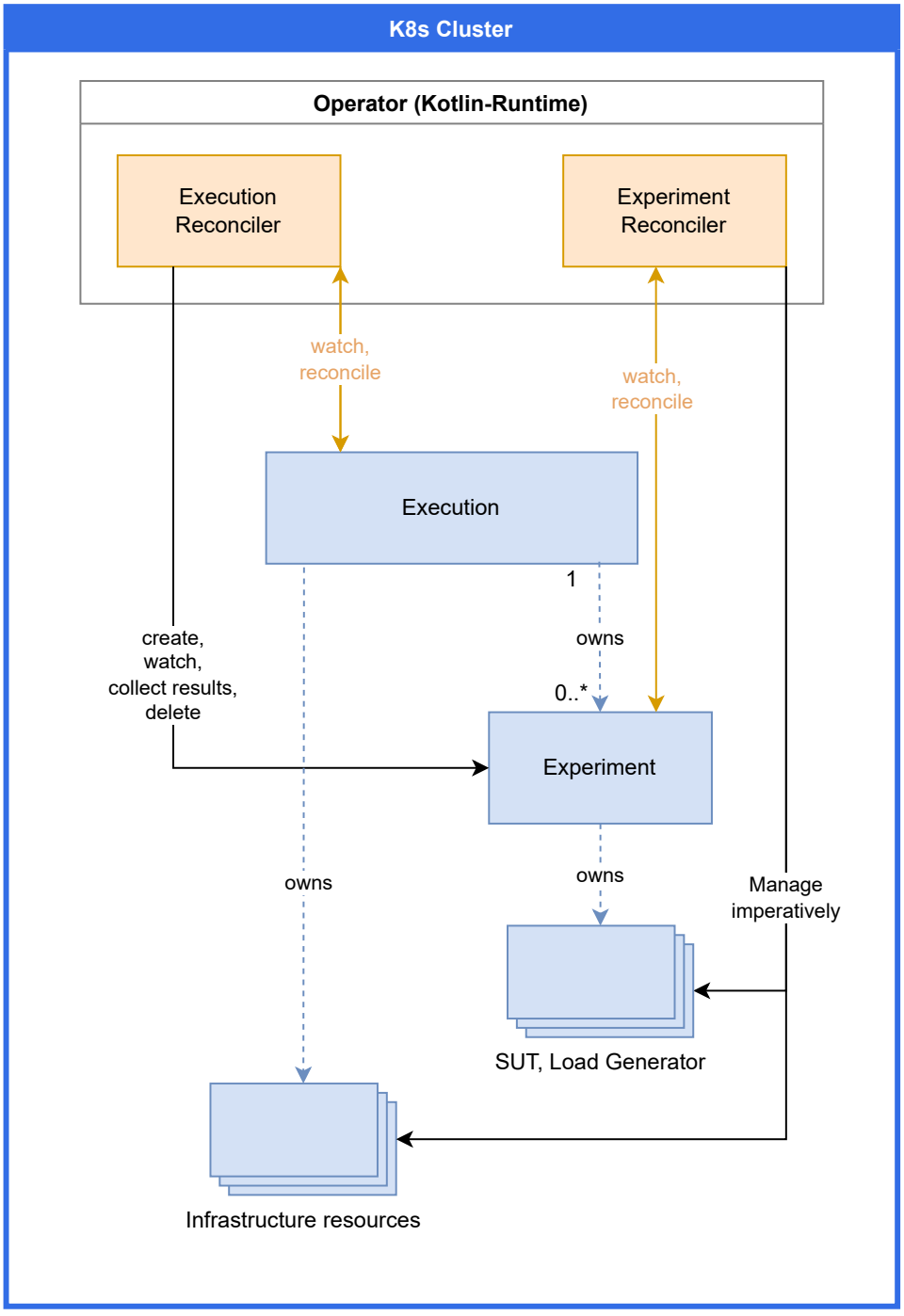


Figure 4.5. Approach 2: Individual Experiments are started by deploying a Kubernetes resource.

Chapter 5

Implementation

Since we devised two architectures for the operator in the previous chapter, we justify our choice of architecture in Section 5.1. Then, we discuss different techniques of modeling the lifecycle of custom Kubernetes resources in Section 5.2. Finally, we describe the lifecycles and implementation of the Kubernetes controllers responsible for managing Theodolite's custom resources; namely the `Benchmark` controller in Section 5.3 and the `Execution` controller in Section 5.4. The section on the `Execution` controller most notably contains a description of the local component responsible for the actual benchmarking logic.

5.1 Implemented Approach

After careful consideration, we chose to implement approach 1.2. We decided against approach 2 for the following reasons:

5.1.1 Why Approach 1 was Chosen Over Approach 2

Approach 2 could certainly result in a more elegant implementation, but most of its advantages have a lower priority for Theodolite, as explained in the following.

Fault tolerance Since spurious operators terminations have been observed to occur rarely during the lifetime of the current operator, this quality attribute has a lower priority. However, it is important to note that this is based on the anecdotal experience of developers and users, and not on a statistical analysis.

Fault-tolerance is especially important for business-critical applications that are required to be available all the time. If such an application does not handle faults well (i.e., is not available for a longer time), this can have severe consequences for the business. By contrast, if a fault causes a restart of the Theodolite operator, the last benchmark has to restarted from the beginning in the worst case.

Apart from this, it is possible to implement a fault-tolerant operator with both approaches, as we will discuss in Section 8.2.1.

Horizontal scalability As discussed in Section 4.5, a horizontally scalable operator only offers the benefit of a reduced restarting time in case of a failure. This might be a

desirable feature for business-critical applications, but is a negligible advantage in our case: The Theodolite operator taking 5 seconds longer to recover from a failure is most likely not very noticeable to the user. Even if multiple instances of the operator could be fully utilized in parallel, it is highly unlikely that this will be necessary, as its workload is quite limited: The Theodolite operator may need to reconcile multiple `Benchmark` and `Execution` resources, but only one of these `Execution` resources will be active at any given time, since running multiple `Executions` in parallel can skew the benchmarking-results.

5.2 Modeling the State and Lifecycle of Kubernetes Resources

5.2.1 State Machines

According to Lamport [2008], state machines are one of the most important techniques for formally describing program behavior and computations. They are used in a plethora of areas in computer science, for example in formal software verification [Yuang 1988], neural networks [Hudson and Manning 2019] and in a distributed context [Schneider 1990]. They offer many advantages, such as providing an easy-to-understand visualization of the system's behavior. Furthermore, state machines can often be directly translated into code.

However, when it comes to modeling the state of custom Kubernetes resources, state machines suffer from two disadvantages: Custom resources are reconciled using level-based logic (as explained in Section 2.2.2), meaning that the control loop does not use the information conveyed by the events it receives, but always uses the current state of the cluster as the basis of its logic. This already contradicts the very nature of state machines, where the semantic of incoming events determines the next state of the system.

Furthermore, the state of a custom resource often depends on multiple conditions, such as the deployment status of a list of other resources in the cluster. It is quite easy to run into a "state explosion"; a problem that is also very prominent in model checking [Park and Kwon 2006]. For example, consider a custom resource that is in the "Ready" state if and only if four conditions are true. Since any permutation of these conditions can occur, the state machine visualization would already consist of $2^4 = 16$ states.

Defining a state for every possible situation is often impractical. Instead, we opted for using *conditions*, which are described in the following section. It is worth noting that the decision against state machines does not seem to be shared across the entire Kubernetes Community. For example, there is a library that generates Kubernetes Operators from state machines.[1]

[1] `https://github.com/krator-rs/krator`

5.2.2 Conditions

The official documentation on the Kubernetes API-Conventions recommends representing the status of a Kubernetes resource by a list of *conditions* [The Kubernetes Authors 2022b]. A condition is a boolean-statement (with additional metadata) about a property related to the custom resource (e.g., whether all sub-resources were found in the cluster). Since conditions are only observations of the current cluster state, they can be related more directly to level-based controller implementations than state machines can. Additionally, since state machines usually have a fixed number of states, adding states in the future can often cause a breaking change in the outward-facing API. In contrast, new conditions can usually be added without affecting existing systems. Another advantage of conditions is that they can be very easily consumed by monitoring systems such as Prometheus, which favors observability.

For these reasons, we used conditions as the main representation of the resource's current state.

5.2.3 Phases

Conditions can be complemented by **phases**, which are a high-level aggregation of the conditions. Though phases are similar to states (in that they are also mutually exclusive), they are not intended to be consumed programmatically. Their only purpose is to provide the user with a quick summary of the resource's current status.

5.3 The `Benchmark` Controller

`Benchmarks` only hold general information about the benchmark, but do not yet entail the execution of any actual benchmarking logic. Therefore, the `Benchmark` controller is comparatively simple.

5.3.1 Reconciliation Triggers and Informers

Benchmark reconciliation should be triggered in two cases: Either the Specification (`.spec` field) of the `Benchmark` is modified, or a resource referenced by the benchmark is created, updated or deleted. The former case is automatically handled by the Java Operator SDK, but the latter needs manual configuration. The Java Operator SDK allows the configuration of reconciler triggers by letting the user register different *event sources* (as explained in Section 2.4). For the benchmark reconciler, the `InformerEventSource` is suitable. As the name suggests, it uses *informers* (see Section 2.2.4) to cache the relevant `ConfigMaps` and notify the reconciler when they change.

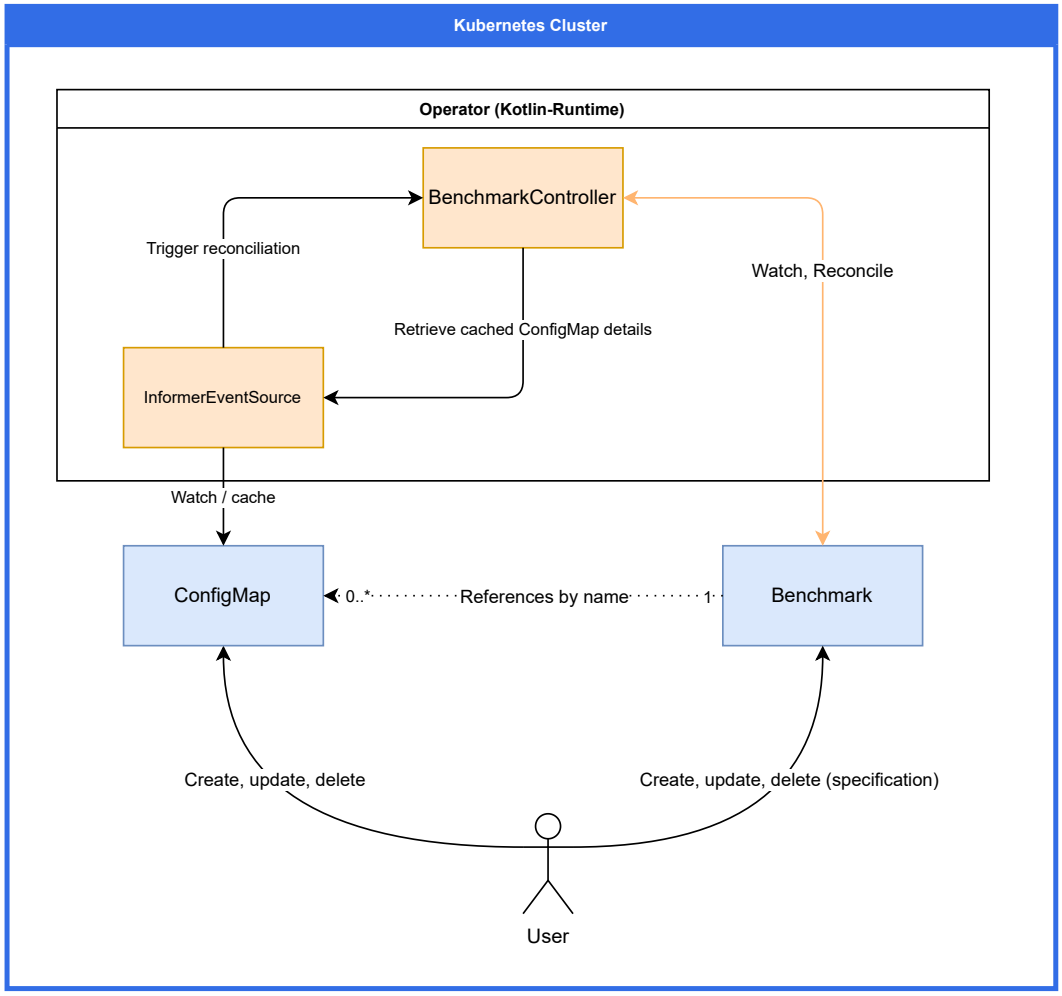


Figure 5.1. An overview over the `BenchmarkController` and its reconciliation triggers.

5.3.2 Benchmark Lifecycle

Although Benchmarks are conceptually stateless [Henning and Hasselbring 2022], their status field still contains information about resources they require. They can be best described by the following conditions.

Benchmark Conditions

Table 5.1. The conditions of the Benchmark custom resource. A condition is monotonic if the condition cannot become false after it has become true. It is to be noted that the InternalError condition is not monotonic, implying that benchmarks can recover from errors, if the error is not thrown again in the next reconciliation.

Condition	Description	Monotonic
Initialized	The resource was reconciled at least once.	Yes
ReferencedConfigMapsFound	All ConfigMaps referenced by this Benchmark were found in the cluster.	No
ReferencedFilesFound	All files referenced by this Benchmark were either found in one of the referenced ConfigMaps or locally.	No
Ready	Ready $\Longleftrightarrow$ Initialized $\wedge$ ReferencedConfigMapsFound $\wedge$ ReferencedFilesFound	No
InternalError	An unexpected error has occurred during the last reconciliation.	No

Benchmark Phases

The Benchmark conditions are aggregated into the following three phases:

Table 5.2. The high-level phases of a benchmark and how they are computed from the conditions.

Phase	Logical formula(based on conditions)	Description
Pending	$\neg Ready \wedge \neg InternalError$	The Benchmark is waiting for the deployment of additional resources it requires (i.e., ConfigMaps, Files)
Ready	$Ready \wedge \neg InternalError$	The Ready-Condition is true (all required resources have been deployed).
Error	$InternalError$	An unexpected error has occurred.

5.4 The `Execution` Controller

This section is structured after the three main tasks of the `Execution` controller, which are as follows:

1. Keep the `.status` field of the `Execution` custom resource up-to-date.
2. Manage the execution of the benchmarking logic (via the `ExecutionRunner` component).
3. Make the state of the `ExecutionRunner` observable to the user.

5.4.1 The Reconciliation Process

Since we decided for approach 1 in Section 4.3, the execution controller will need to manage a component that asynchronously executes the benchmarking logic in a local context (a `ExecutionRunner`). Additionally, it needs to ensure that there exists only one such component per `Execution`, and that the component is terminated when the `Execution` is deleted.

Hence, the `Execution` control loop will generally be structured as follows:

1. Read the current state of the cluster,
2. Check whether the benchmarking logic should be executing right now,
3. Manage the `ExecutionRunner` for this execution to reflect this,
4. Update the `Execution`'s status accordingly.

Step 3 is of particular interest, since it involves managing the local state, as opposed to the cluster state. Therefore, we discuss it in more detail in the next subsection.

5.4.2 The `ExecutionRunner` Component

An `ExecutionRunner` is a unit responsible for locally executing the benchmarking logic. There should be at most one `ExecutionRunner` per `Execution` at any given time. This property is ensured by forcing access through the `ExecutionRunnerRepository` (see Figure 5.2).

Controlling the `ExecutionRunner`: Imperative vs. Declarative Approach

The `ExecutionReconciler` needs to ensure that the local `ExecutionRunner` is in the correct state. It is possible to implement this imperatively (the reconciler starts, updates and stops the `ExecutionRunner`, depending on the cluster state) or declaratively (the reconciler communicates a desired state of the `ExecutionRunner` (i.e., running(benchmark, execution), not running), which the latter then converges to on its own). Since the `ExecutionRunner` is a separate component, it is possible to implement both approaches.

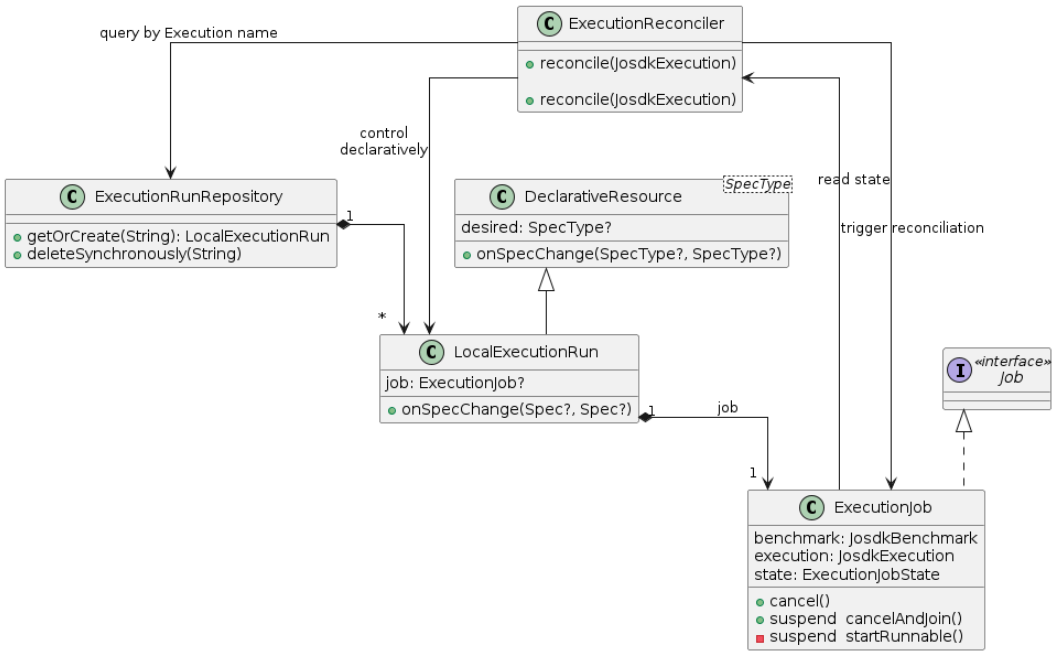


Figure 5.2. A class diagram presenting the relevant classes around the `ExecutionReconciler`. `ExecutionJobs` are ephemeral objects that only exist during the actual execution of the benchmarking logic.

`isExecutionReady`	State of the runner	Consequence
false	None	Do nothing
false	Running	Stop runner
true	None	Start runner
true	Running	Do nothing
X	Stale[a]	Update and restart the runner
X	Error	Restart runner if remaining retries > 0
X	Succeeded	Do nothing

[a] The runner is stale if the `Benchmark` or `Execution` have been updated since the runner was started.

Conceptually, the desired behavior of the `ExecutionReconciler` can most easily be described in an imperative way:

We initially implemented the imperative approach, but later decided to switch to the declarative approach, because it yields a better separation of concerns: The `Execution` reconciler does not need to implement any lifecycle-related logic for the `ExecutionRunner`. This results in a weaker coupling between the reconciler and the `ExecutionRunner`, which entails the usual advantages of decoupling, mainly easier unit testing and easier replacement. For example, it is possible to replace the `ExecutionRunner` by a dedicated Kubernetes resource, should this be necessary in the future.

Listing 5.1. By using a declarative approach, the `ExecutionReconciler` does not need to implement any logic related to the `ExecutionRunner` itself.

```
1    val shouldExecutionRun = /* Determine this based on the cluster state */
2
3    if (shouldExecutionRun) {
4        localRunner.spec = ExecutionRunnerSpec(benchmark, execution)
5    } else {
6        localRunner.spec = null
7    }
```

The `ExecutionRunner` Lifecycle

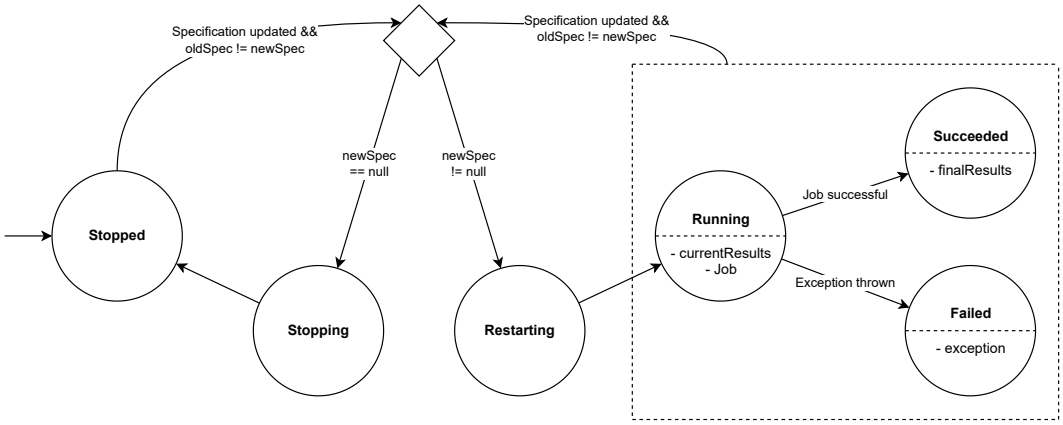


Figure 5.3. The lifecycle of the unit responsible for executing the benchmarking logic.

5.4.3 The Execution Lifecycle

Since the lifecycle of the Execution custom resource is conceptually very similar to the lifecycle of the ExecutionRunner (Figure 5.3), we will only shortly present its conditions.

Condition	Description
isAssociatedBenchmarkReady	Whether the Benchmark referenced by this Execution was found and is ready.
isExecutionNextInImplicitQueue	Whether this Execution is the next to run.
isReady	isAssociatedBenchmarkReady $\wedge$ isExecutionNextInImplicitQueue
isActive	Whether the local runner is currently running.
isSucceeded	Whether the local runner has successfully finished the execution logic for the current version of both the Benchmark and Execution
isFailed	Whether the local runner has failed.

Chapter 6

Performance Evaluation

We evaluate the performance of the reengineered operator by comparing it to the current operator. For this, we let both operators execute the same workload (in the form of a Theodolite `Benchmark` and `Execution`) and measure the results in terms of CPU usage and load on the Kubernetes API.

6.1 Methodology

6.1.1 Hard- and Software Setup

For all experiments, a Google Cloud Engine Kubernetes cluster[1] with the following specifications was used:

Machine type `e2-standard-4` (4 vCPUs, 16 GB memory)

Number of nodes 1 (to avoid non-deterministic behavior)

Operating system Container-optimised OS with `containerd` (`cos_containerd`)

Kubernetes control plane version `1.24.3-gke.2100`

Boot disk (per node) 96 GB, Balanced persistent disk

6.1.2 Operator Workload

Since we are only evaluating the operator performance, the underlying workload needs to fulfill two requirements:

- It should not impose a high load on the cluster, to avoid throttling of the operator container.
- It should require exactly the same steps from the operator as a regular benchmark would.

To fulfill these requirements, we tested the operator by letting it execute the following Theodolite `Benchmark`.

[1] `https://cloud.google.com/kubernetes-engine`

Table 6.1. The benchmark configuration used for the performance evaluation.

Component	Description
System Under Test (SUT)	An `nginx`-webserver serving a simple static website.
Load Generator	A simple script that sends a variable amount of HTTP requests to the server.
Search Strategy	`FullSearchStrategy`
SLO	A mock-SLO that is always fulfilled.
Amount of experiments	16
Duration per experiment	120 seconds
Repetitions per experiment	1

The detailed deployment files for the `Benchmark` and `Execution`, as well as scripts to reproduce the experiments can be found in the artifact repository for this thesis [Mertens 2022].

6.1.3 Resource units in Kubernetes

Cumulative CPU time in seconds

The CPU usage of applications on Kubernetes is measured in CPU-Seconds. For example, a process using 1 CPU second, is equivalent to one CPU core executing only the process for 1 second (scheduling overhead is not included in this metric). An advantage of this metric is that it is an absolute measurement, in that it is independent of the amount of cores of the system. Under UNIX systems, CPU-Seconds for each process are exposed by the Kernel[2].

Kubernetes CPU units

According to the Kubernetes documentation [The Kubernetes Authors 2022a], the CPU resources of a Kubernetes cluster are measured in *cpu* units[3].

The *cpu* unit at a point in time t is calculated by the rate of change in cumulative CPU seconds in a variable time window before t [The Kubernetes Authors 2022d].

We be comparing the CPU usage of the two operators using the *cpu* unit, because it allows for an accurate comparison of CPU usage, assuming that the same CPU type is used for both experiments.

[2] `https://pubs.opengroup.org/onlinepubs/9699919799/functions/clock.html`

[3] `https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#resource-units-in-kubernetes`

6.1.4 Considered Metric Sources

We initially considered the following three metric sources:

- Metrics from the monitoring system Prometheus [The Prometheus Authors 2022]
- Google Cloud Monitoring metrics (more specifically, the `system` type metrics[4], because they provided a higher sample rate for the CPU-based metrics)
- Manually polling the Kubernetes Metric API (which is internally used by the `kubectl top` command).

6.1.5 Explorative Pre-Study: Evaluation of CPU Metric Sources

All three of our considered metric sources offer information about the CPU usage of specific pods. They are all based on the amount of cumulative CPU seconds (see Section 6.1.3), but differ in how they calculate the current CPU usage.

We evaluated them by measuring and comparing their outputs in a set time range. In the time range, the reengineered Theodolite operator was deployed, and an `Execution` was started, to simulate a realistic CPU usage.

The following metrics were compared:

Google Cloud Metrics

The following mql[5] query was used:

```
fetch k8s_container
  | metric 'kubernetes.io/container/cpu/core_usage_time'
  | filter (resource.container_name == 'theodolite')
  | align rate(15s)
  | every 15s
```

Prometheus

The CPU-usage is calculated by the rate of change in cumulative CPU over a sliding time window of 15 seconds, as described by the following PromQL[6] query.

```
rate(container_cpu_usage_seconds_total{pod=~"theodolite.*", container="theodolite"}[15s])
```

[4] https://cloud.google.com/stackdriver/docs/solutions/gke/managing-metrics#system-metrics

[5] https://cloud.google.com/monitoring/mql

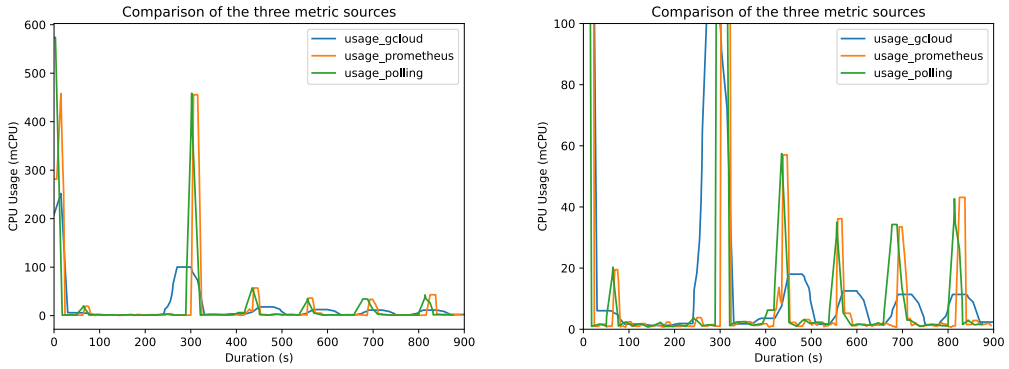
[6] https://prometheus.io/docs/prometheus/latest/querying/basics/

Directly polling the Metrics-API

Data points were obtained by querying the Kubernetes Metrics Endpoint (`metrics.k8s.io/v1beta1/default/pods`) for the CPU and memory usage of the operator-container. The metric server was configured to use a `metric-resolution` of 15 seconds, which results in a new data point becoming observable every 15 seconds.

An example result contains the following data:

```
"name":"theodolite",
"usage":{"cpu":"1100981n", "memory":"203724Ki"},
"timestamp":"2022-09-28T07:52:21Z",
"window":"12s"
```



(a) The full graph.

(b) With a limited y-axis to exclude extreme peaks.

Figure 6.1. Comparison of the three metric sources for CPU usage. Data points were aligned by the timestamps reported by each respective metric source.

The results are visualized in Figure 6.1. All three graphs show a similar pattern of peaks and valleys, where a peak corresponds to the operator reconciling a custom resource. While the data points from Prometheus and from the Metrics-API seem to be very similar, the GCloud metrics exhibit much shorter, wider peaks. We suspect that this is caused by the GCloud metrics being calculated as the rate of change over a longer time window than the other two metrics. For this reason, we deemed the GCloud metrics unsuitable for this evaluation.

We suspect that both Prometheus and the Metrics-API are based on the same data source, i.e., the metrics server [The Kubernetes Authors 2022d]

Based on the comparison, we decided to use the Kubernetes Metrics API as our CPU metric source.

6.1.6 Measuring the Kubernetes API Server Load

Google Cloud - hosted Kubernetes clusters, such as the one used for this evaluation, do not directly expose information about the Kubernetes API-Server to the user. Hence, we had to consider other metrics that could be used to infer the load on the Kubernetes API-Server.

The following PromQL query was used:

```
sum(rate(apiserver_response_sizes_sum{group!~"monitoring.*"}[5m])) +
sum(rate(apiserver_watch_events_sizes_sum{group!~"monitoring.*"}[5m]))
```

We use the API response sizes as a heuristic for the load on the API server. This query sums the rate of change in the size of the responses to all requests (except for responses related to monitoring resources).

The metric also includes requests from sources other than the Theodolite operator, because it is not possible to filter requests by their origin. However, our evaluation cluster only had the Theodolite operator and monitoring-resources (such as Prometheus) installed. Requests to monitoring-resources were excluded from the query, so the amount of unrelated requests should be relatively constant. Since we are only interested in comparing the two operators, we can accept a constant amount of noise.

6.1.7 Evaluation Experiments

To compare the performance of both operators, we sequentially execute the following experiments for both operators:

Warm up phase The operator is running on the Java Virtual Machine (JVM). Applications on the JVM are known to be complicated to evaluate due to the JVM's non-deterministic behavior (caused, for example, by the Just-In-Time compiler) [Georges et al. 2007], [Georges et al. 2008]. Hence, before running any other experiments, we will execute a 5-minute warm up phase, to reduce the influence of the JVM-startup on the results.

Idle with custom resources The operator has been running for 5 minutes, a `Benchmark` and an `Execution` have been deployed, but the `Execution` is not yet ready to run.

Running The operator has been running for 10 minutes, the `Benchmark` and `Execution` have been deployed, and the `Execution` described in Section 6.1.2 is being run by the operator.

6.2 Results and Discussion

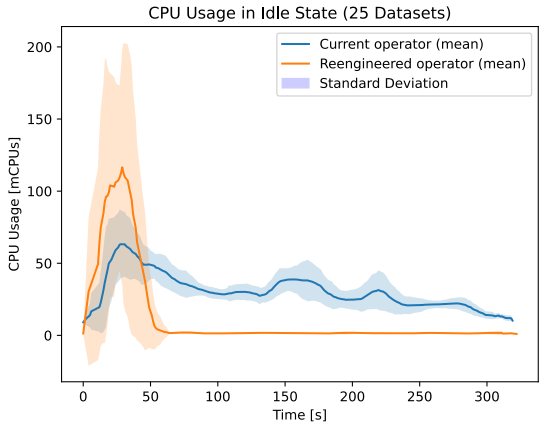


Figure 6.2. CPU usage after the warm-up phase. A `Benchmark` and an `Execution` have been deployed, but the `Execution` is not yet ready to run. The initial peak is most likely caused by the first execution reconciliation. After that, the reengineered operator is using less CPU than the original operator, because no reconciliation is necessary. We calculated an average reduction of CPU usage by 73.35%.

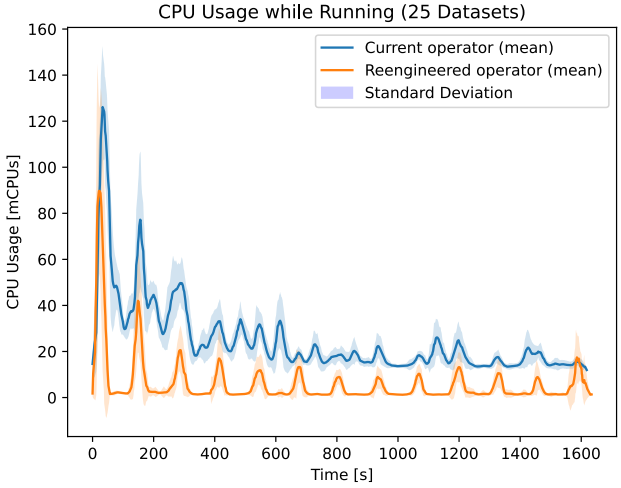


Figure 6.3. CPU usage when running an execution. The current operator shows a small overhead due to its use of polling. The peaks of the reengineered operator reflect the reconciliations taking place after each SLO-Experiment. We calculated an average reduction of CPU usage by 79.41%.

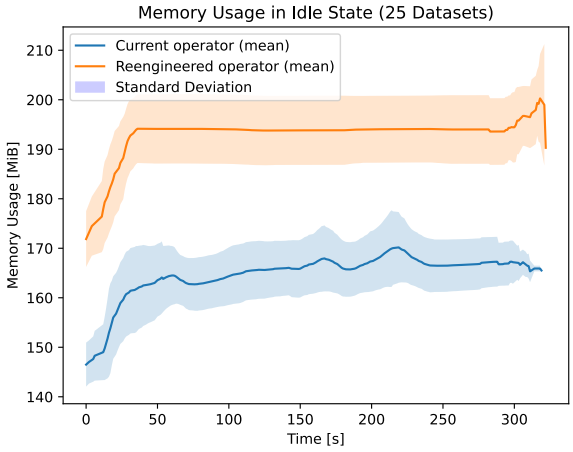


Figure 6.4. Memory usage after the warm-up phase. We observe an increase in container memory consumption in the reengineered operator when running an execution. We calculated an average reduction of memory usage by 11.31%.

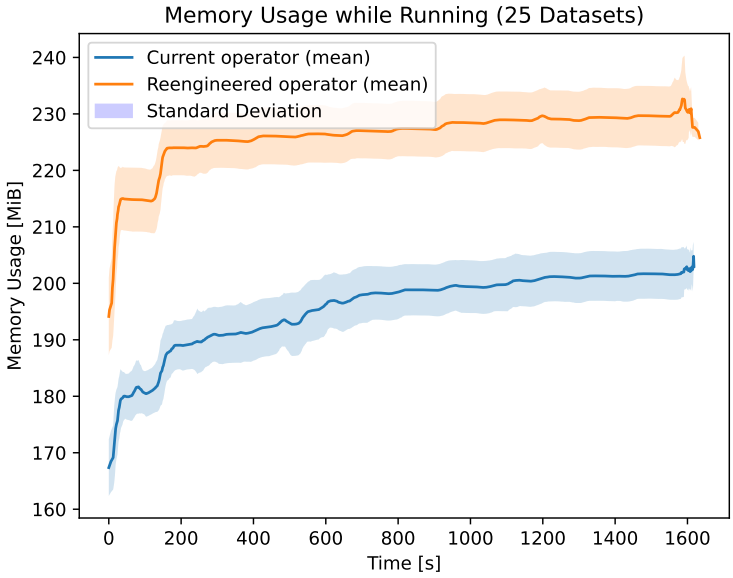


Figure 6.5. We observe an increase in container memory consumption in the reengineered operator when running an `execution`. We calculated an average increase of memory usage by 27.74%.

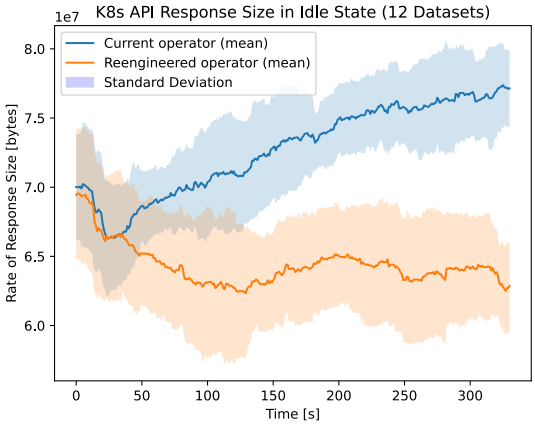


Figure 6.6. API response sizes before the execution has started. The reengineered operator seems to cause smaller API response sizes than the current operator, however, the standard deviation is quite high, so we can only observe a general trend. We calculated an average reduction of API response size by 16.00%.

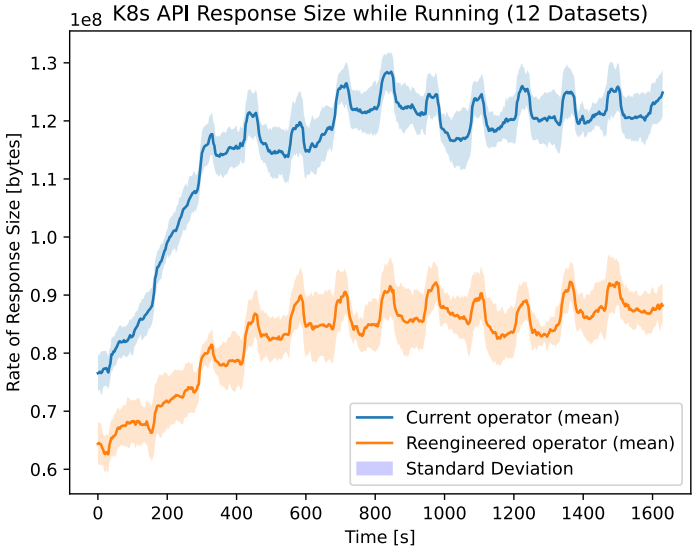


Figure 6.7. API response sizes when running an execution. The standard deviation is still quite high (note the larger range on the y-axis). However, we can observe a clear improvement in the reengineered operator. We calculated an average reduction of API response size by 16.00%.

6.3 Threats to Validity

6.3.1 Internal validity

Calculation of the Rate of Change

As discussed previously, the CPU usage at a point in time is calculated by the rate of change of the cumulative CPU seconds in a (sliding) time window. In the results obtained from the Kubernetes Metrics API, this time window is not fixed, but seems to equal the time between two consecutive measurements of CPU seconds. In our measurements, we observed time window sizes between 12 and 20 seconds, because we configured the Kubernetes API to scrape metrics every 15 seconds[7].

This technique of calculating the CPU usage is not very precise. If the time window is too large, sudden peaks in CPU usage will be smoothed out[8] (i.e., become "wider and flatter"), as seen in Section 6.1.5. However, since we chose a comparatively small time window, this effect should not be very pronounced in our measurements.

API Load Metric

To indirectly measure the load on the Kubernetes API server, we use the API response size as a proxy. As mentioned before, we assume that the size of the responses is roughly proportional to the amount of data that the API server has to process. This assumption must not necessarily be true, for example if the API server caches data.

As discussed in the same section, we also accept a certain amount of noise in the measurements (which can be observed in the standard deviation of the measurements in Figure 6.6 and Figure 6.7). We try to counteract this by executing multiple experiments and averaging the results. Nevertheless, this metric only allows for insight into the general trend of the API load,

6.3.2 External validity

The `Benchmark` and `Execution` used in the evaluation are very simple and are thus not representative of real-world use cases. However, this should not impair the validity of the results regarding the operators, because the actions taken by the operator are very similar regardless of the underlying `Benchmark` and `Execution`.

[7] Values under 15s are discouraged: https://github.com/kubernetes-sigs/metrics-server/blob/master/FAQ.md#how-often-metrics-are-scraped

[8] This stems from the fact that this type of metric is usually used to control autoscaling, where the smoothing of smaller peaks is desirable.

Chapter 7

Related Work

7.1 Theodolite

Theodolite was initially presented with a strong focus on the use case of benchmarking stream processing engines ([Henning and Hasselbring 2021]). Its Kubernetes operator architecture was later explained in detail [Henning et al. 2021]. In a more recently published article, Henning and Hasselbring [2022] generally explain how Theodolite's scalability metrics and measurement work, and make recommendations for choosing configuration parameters (e.g., the length of experiments), based on an experimental evaluation.

7.2 Scalability and Cloud Benchmarking

Choochotkaew et al. [2022] very recently presented AutoDECK[1], a framework for performance evaluation and tuning of Kubernetes native applications. The authors name fault tolerance of the AutoDECK operator as an advantage over other Kubernetes benchmarking tools (among them Theodolite). As another advantage, they mention the tools support for declaratively defined benchmark-infrastructure (e.g., a Kubernetes operator used by all benchmark resources), which Theodolite supports as well (through *infrastructure* resources). AutoDECK and Theodolite seem to share a similar architecture (for example, both pilot the benchmarking process using a Kubernetes operator). Apart from that, the two projects differ in their approach, since performance evaluation and scalability evaluation are fundamentally different objectives.

7.3 Kubernetes Operators

The Kubernetes Operator Framework[2] provides open-source tools and infrastructure related to Kubernetes operators.

Most notably, it contains the `operator-sdk`[3], a toolkit for developing operators in the Go programming language[4]. Hausenblas and Schimanski [2019] provide a practical guide

[1] Not to be confused with the software company *Autodesk*

[2] `https://github.com/operator-framework`

[3] `https://github.com/operator-framework/operator-sdk`

[4] `https://go.dev/`

on how to develop operators using the `operator-sdk`. Dobies and Wood [2020] focus slightly more on general concepts of the operator pattern and Kubernetes, but also use the `operator-sdk` to showcase the development process of a Kubernetes operator.

The Operator Framework also defines a maturity model[5] for Kubernetes operators, which rates Kubernetes operators on a five-level scale, based on their functionality (e.g., whether the operator supports the exposure of metrics and automatic scaling of the operated components). Duan et al. [2021] have presented the maturity model in more detail, implemented and evaluated an operator for a sample application that they claim to be in the highest maturity level.

7.4 Kubernetes Operators Written with the Java Operator SDK

Several operators, have been developed with the Java Operator SDK. In this section, we present some of them.

The open source distributed stream processing engine Apache Flink [Katsifodimos and Schelter 2016] uses an operator[6] implemented with the Java Operator SDK to manage its lifecycle.

Further examples of operators written with the Java Operator SDK include the following:

- An operator for the identity management software keycloak[7].
- An operator for the Oracle WebLogic Server [8].
- An operator written for the data analytics engine Apache Spark, which simplifies the management of Spark clusters on Kubernetes[9].

[5] `https://operatorframework.io/operator-capabilities`
[6] `https://github.com/apache/flink-kubernetes-operator`
[7] `https://github.com/keycloak/keycloak/tree/main/operator`
[8] `https://github.com/oracle/weblogic-kubernetes-operator`
[9] `https://github.com/radanalyticsio/spark-operator`

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this work, we have identified shortcomings of Theodolite's current operator by analyzing its code and architecture, as defined in Goal G1. The main shortcomings were related to efficiency (due to frequent polling of the Kubernetes API), observability and understandability and fault tolerance.

We have designed two architectures for Theodolite's operator, taking the elicited shortcomings into account. The first one contains a local component that sequentially executes the benchmarking logic, while the second one divides the benchmarking logic into individual parts. The former is significantly less complex, while the latter is essentially stateless, which makes the operator more scalable and fault-tolerant. We identified that optimizing Theodolite's operator for scalability and fault tolerance would yield very little benefit, compared to the increased complexity. Hence, we implemented the first architecture using the Java Operator SDK. This corresponds to Goal G2.

Finally, we evaluated whether the reengineered operator reduces the CPU usage and the load on the Kubernetes API, by comparing it to the current operator, as planned in Goal G3. For this, we first evaluated possible sources of metrics for the CPU usage of the operator. We measured a reduction of CPU usage and API response sizes, and an increase of memory usage in the reengineered operator.

8.2 Future Work

We have presented two approaches to the problem of modeling the benchmarking logic in a Kubernetes operator in Section 4.3, out of which the first approach was chosen for the reasons listed in Section 5.1.

However, it would be interesting to see whether the second approach is feasible in practice. Since the second approach makes the operator essentially stateless (by storing its state in the Kubernetes API), the following steps (among others) would most likely be necessary to implement it:

8.2.1 Making `SearchStrategies` Stateless by Adding a "Replay" Functionality

As discussed in Section 3.2.1, `SearchStrategies` make up a large portion of the operators local state. To work with a stateless operator, they could be modified to offer a function with a signature akin to this:

```
nextExperimentToRun ::
    loadValues, resourceValues, metric, previousExperimentResults
    -> (loadValue, resourceValue)
```

This function could repeatedly be called in the reconciliation loop, to determine the experiment to run next.

As explained in Section 3.2.1, making `SearchStrategies` entirely stateless is not possible (consider, for example, the `BinarySearchStrategy`, which depends on the call stack as internal state). Hence, to implement this function for all `SearchStrategies`, the internal runtime-state of the strategy would need to be reconstructed (*replayed*) each time it is called, using the experiment results collected so far. This replay mechanism could look something like this:

1. Start the `SearchStrategy`.

2. As long the strategy demands the execution of an experiment that has already been run: Feed the previously determined result into the strategy.

3. Once the strategy demands the execution of an experiment that has not yet been run: Return the demanded experiment.

This functionality would also be useful for approach 1: The replay-mechanism could be used to restart the sequential benchmarking logic from the last saved state, if the operator detects that it has crashed.

8.2.2 `ExecutionReconciler` for Approach 2

The ExecutionReconciler would be the main component of the operator in approach 2. A possible implementation is depicted in Figure 8.1. It remains questionable whether the effort required to implement this approach is worth the limited benefits.

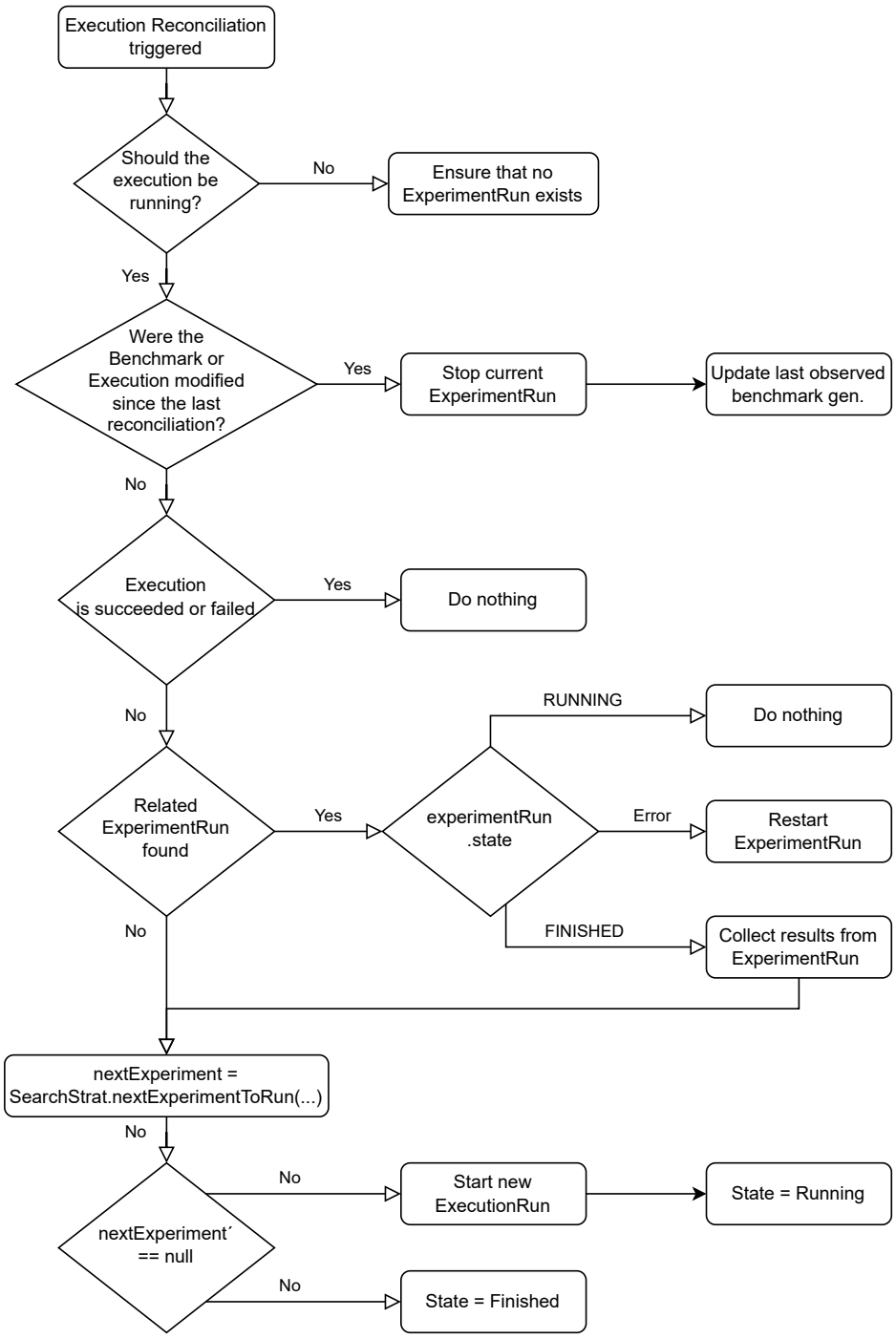


Figure 8.1. A high level sketch of how the `ExecutionReconciler` could look for approach 2.

Bibliography

[Chikofsky and Cross 1990] E. Chikofsky and J. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software* 7.1 (1990), pages 13–17. DOI: 10.1109/52.43044. (Cited on page 3)

[Choochotkaew et al. 2022] S. Choochotkaew, T. Chiba, S. Trent, T. Yoshimura, and M. Amaral. AutoDECK: automated declarative performance evaluation and tuning framework on kubernetes. In: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. IEEE, July 2022. DOI: 10.1109/cloud55607.2022.00053. (Cited on page 47)

[Dobies and Wood 2020] J. Dobies and J. Wood. *Kubernetes Operators*. Sebastopol, CA: O'Reilly Media, Feb. 2020. URL: https://www.oreilly.com/library/view/kubernetes-operators/9781492048039. (Cited on pages 1, 5, 48)

[Duan et al. 2021] R. Duan, F. Zhang, and S. U. Khan. A case study on five maturity levels of a kubernetes operator. In: *2021 IEEE Cloud Summit (Cloud Summit)*. IEEE, Oct. 2021. DOI: 10.1109/ieeecloudsummit52029.2021.00008. (Cited on page 48)

[Georges et al. 2007] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pages 57–76. DOI: 10.1145/1297027.1297033. (Cited on page 41)

[Georges et al. 2008] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. *ACM SIGPLAN Notices* 43.10 (Oct. 2008), pages 367–384. DOI: 10.1145/1449955.1449794. (Cited on page 41)

[Hausenblas and Schimanski 2019] M. Hausenblas and S. Schimanski. *Programming Kubernetes*. Sebastopol, CA: O'Reilly Media, July 2019. (Cited on pages 4, 5, 17, 47)

[Henning and Hasselbring 2021] S. Henning and W. Hasselbring. Theodolite: scalability benchmarking of distributed stream processing engines in microservice architectures. *Big Data Research* 25 (July 2021), page 100209. DOI: 10.1016/j.bdr.2021.100209. (Cited on pages 1, 47)

[Henning and Hasselbring 2022] S. Henning and W. Hasselbring. A configurable method for benchmarking scalability of cloud-native applications. *Empirical Software Engineering* 27.6 (Aug. 2022), page 143. DOI: 10.1007/s10664-022-10162-1. (Cited on pages 7–9, 15, 31, 47)

[Henning et al. 2021] S. Henning, B. Wetzel, and W. Hasselbring. Reproducible benchmarking of cloud-native applications with the Kubernetes operator pattern. In: *Symposium on Software Performance 2021*. CEUR Worshop Proceedings. Nov. 2021. URL: https://oceanrep.geomar.de/id/eprint/54582/. (Cited on pages 7, 47)

[Hudson and Manning 2019] D. Hudson and C. D. Manning. Learning by abstraction: the neural state machine. *Advances in Neural Information Processing Systems* 32 (2019). (Cited on page 28)

[Katsifodimos and Schelter 2016] A. Katsifodimos and S. Schelter. Apache flink: stream analytics at scale. In: *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*. IEEE, Apr. 2016. DOI: 10.1109/ic2ew.2016.56. (Cited on page 48)

[Lamport 2008] L. Lamport. Computation and state machines (2008). URL: https://www.microsoft.com/en-us/research/publication/2016/12/Computation-and-State-Machines.pdf. (Cited on page 28)

[Mertens 2022] L. Mertens. *Thesis artifacts for: reengineering theodolite with the java operator sdk*. Sept. 2022. DOI: 10.5281/zenodo.7121210. (Cited on page 38)

[Park and Kwon 2006] S. Park and G. Kwon. Avoidance of state explosion using dependency analysis in model checking control flow model. In: *Computational Science and Its Applications - ICCSA 2006*. Edited by M. L. Gavrilova, O. Gervasi, V. Kumar, C. J. K. Tan, D. Taniar, A. Laganá, Y. Mun, and H. Choo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pages 905–911. (Cited on page 28)

[Red Hat 2022] Red Hat. *What are Red Hat OpenShift operators?* 2022. URL: https://www.redhat.com/en/technologies/cloud-computing/openshift/what-are-openshift-operators (visited on 08/17/2022). (Cited on page 5)

[Red Hat, Container Solutions 2022] Red Hat, Container Solutions. *Java Operator SDK*. 2022. URL: https://javaoperatorsdk.io/. (Cited on pages 1, 8, 17, 19, 20)

[Schneider 1990] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.* 22.4 (Dec. 1990), pages 299–319. DOI: 10.1145/98163.98167. (Cited on page 28)

[The Kubernetes Authors 2018] The Kubernetes Authors. *Client Go controller interaction*. 2018. URL: https://github.com/kubernetes/sample-controller/blob/50026bca24dec1e5168df3b671b97a5d97098f22/docs/images/client-go-controller-interaction.jpeg (visited on 08/05/2022). (Cited on page 6)

[The Kubernetes Authors 2021] The Kubernetes Authors. *Kubernetes documentation: controllers*. 2021. URL: https://kubernetes.io/docs/concepts/architecture/controller/ (visited on 08/16/2022). (Cited on page 4)

[The Kubernetes Authors 2022a] The Kubernetes Authors. *Kubernetes API concepts*. 2022. URL: https://kubernetes.io/docs/reference/using-api/api-concepts/#efficient-detection-of-changes (visited on 08/05/2022). (Cited on pages 5, 38)

[The Kubernetes Authors 2022b] The Kubernetes Authors. *Kubernetes API conventions*. 2022. URL: https://github.com/kubernetes/community/blob/e1bef6b4c1140971c1dde6b6f8b42c57f6309990/contributors/devel/sig-architecture/api-conventions.md (visited on 09/18/2022). (Cited on page 29)

[The Kubernetes Authors 2022c] The Kubernetes Authors. *Kubernetes documentation: ReplicaSet*. 2022. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#when-to-use-a-replicaset (visited on 09/05/2022). (Cited on page 22)

[The Kubernetes Authors 2022d] The Kubernetes Authors. *Resource metrics pipeline*. June 2022. URL: https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/ (visited on 09/28/2022). (Cited on pages 38, 40)

[The Kubernetes Authors 2022e] The Kubernetes Authors. *What is Kubernetes?* 2022. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 05/11/2022). (Cited on page 4)

[The Prometheus Authors 2022] The Prometheus Authors. *What is Prometheus?* 2022. URL: https://prometheus.io/docs/introduction/overview/ (visited on 09/20/2022). (Cited on page 39)

[V. Kistowski et al. 2015] J. V. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao. How to build a benchmark. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: Association for Computing Machinery, 2015, pages 333–336. DOI: 10.1145/2668930.2688819. (Cited on page 1)

[Yuang 1988] M. Yuang. Survey of protocol verification techniques based on finite state machine models. In: *1988 Computer Networking Symposium*. Los Alamitos, CA, USA: IEEE Computer Society, 1988, pages 164–172. DOI: 10.1109/CNS.1988.4993. (Cited on page 28)