**Reproducible Benchmarking of Cloud-Native Applications With the Kubernetes Operator Pattern**

**1. Introduction**

Along with its growing adoption in industry, “cloud-native” software systems and their performance are increasingly studied in research.

Cloud-native applications typically consist of several processes, running in isolated containers. They require application-level configurations passed to these containers, comprehensive descriptions of internal and external network interconnection, and integration of different storage systems. These and other aspects make the deployment and operation of cloud-native applications a rather complex task and, hence, also complicate benchmarking.

On the other hand, several tools emerged in the cloud-native community for enhancing the deployment and operation. For example, Kubernetes is nowadays the de-facto standard for declarative orchestration. We propose to adopt established cloud-native technologies along with the Operator Pattern for designing cloud-native benchmarking tools that particularly support reproducibility.

After a brief introduction of Kubernetes and the Operator Pattern, we present the necessary data model and system architecture for a benchmarking operator.

**2. The Kubernetes Operator Pattern**

Kubernetes is a declarative orchestration tool for cloud-native applications. Users describe the desired state of an application in so-called resources and in a continuous reconciliation loop, Kubernetes adopts the current system to reach the desired state. A typical example of a resource is a Deployment, which describes an application component consisting of container images, number of replicas and others. A common way for users to interact with Kubernetes is by describing resources in YAML files and applying these files to Kubernetes by using the kubectl command line tool.

The Kubernetes Operator Pattern is a recent approach to integrate domain knowledge into Kubernetes’ orchestration process. Implementing this pattern involves two things: First, custom resource definitions (CRDs) define new types of resources that can be managed by the Kubernetes API. Second, a dedicated software component (the actual operator), runs inside the Kubernetes cluster and manages the entire life cycle of what is described by the CRDs. This so called operator interacts with the Kubernetes API and reacts to creation, modification, or removal of custom resources (CRs, instances of CRDs).

**3. Custom Resource Definitions for Benchmarking**

The Operator Pattern combines knowledge of operating Kubernetes with domain knowledge. As the domain at hand is benchmarking, we propose that actors involved in benchmarking describe their operational knowledge in CRDs. In general, we can observe two actors involved in benchmarking:

* The Benchmark Designer has knowledge about operating a system under test (SUT) as well as how the quality to be benchmarked can be measured.
* The Benchmarker executes existing benchmarks.

Based on this distinctions of roles, we propose a data model in which benchmarks and their executions are separated. We envisage individual CRDs for both benchmarks and executions:

* Benchmarks are defined by Benchmark Designers. They describe the SUT as well as the load generation. Additionally, they may describe configurations of the benchmarking method. Benchmarks can be versioned and maintained in public repositories.
* Executions describe a single execution of a Benchmark. They refer to exactly one Benchmark and configure the experimental setup. When deployed to Kubernetes, Executions have a state, which is typically something like Pending, Running, Finished, or Failed if an error occurred.

**4. Benchmarking Operator Architecture**

Standard software components involved in benchmarking are:

* a system under test (SUT),
* a load generator,
* an experiment controller,
* a measurement database,
* an (optional) monitoring component for passive observation.

We propose a benchmarking tool architecture consisting of these components, which applies the Kubernetes Operator.

To execute benchmarks, a Benchmarker registers benchmarks and initiates their executions by applying the corresponding CRs. The operator continuously observes the Kubernetes API for changes in the list of registered executions. It takes on the role of the experiment controller and decides based on a tool-specific policy whether it starts an execution immediately or schedules it to a list of pending executions. Once a registered execution is started, the operator creates and starts the SUT and the load generator via the Kubernetes API as well as performs additional cluster configurations. While executing the benchmark, the operator updates the state of the Execution (e.g., to a state Running).

State and events of an Execution are managed by the Kubernetes API and are used to provide feedback to the Benchmarker. During the entire benchmark execution, Prometheus collects monitoring data from the SUT and the load generator. Once a benchmark execution is finished, the operator stops and removes the SUT and the load generator. In addition, it requests collected monitoring data from Prometheus and analyzes them. Also requesting and analyzing monitoring data during an execution to adjust the SUT and the load generator is possible, which is necessary for certain types of benchmarks.

The operator stores benchmarking results persistently, from where the Benchmarker can access them for offline analysis, archiving, or sharing.

**5. Discussion and Conclusions**

The Kubernetes Operator Pattern allows to manage application-level configuration as part of the orchestration process. Applied to benchmarking cloud-native applications, it allows managing benchmarks and executions directly via the Kubernetes API. Main benefits of a benchmarking operator are:

* Benchmarking tool, benchmark, and execution are clearly separated.
* Benchmarks are defined according to a strict format and published as descriptive files without including any tooling.
* All experimental setup is bundled in one file describing the execution of a benchmark.
* Benchmarks and Executions can be managed via default Kubernetes tools (e.g., kubectl), providing a rich user experience without additional implementation effort.

All in all, we expect building a benchmark tool with the operator pattern facilitates repeatability and, hence, ultimately reproducibility of cloud-native benchmarking studies.

**Воспроизводимое тестирование облачно-ориентированных приложений с использованием паттерна Kubernetes Operator**  
**1. Введение**  
С ростом распространения облачно-ориентированных программных систем в промышленности их производительность всё чаще становится предметом научных исследований.  
Облачно-ориентированные приложения обычно состоят из нескольких процессов, выполняющихся в изолированных контейнерах. Для их работы требуются: конфигурации на уровне приложения, передаваемые в контейнеры; описание внутренних и внешних сетевых взаимодействий; интеграция различных систем хранения данных. Эти аспекты усложняют развёртывание, эксплуатацию и, как следствие, тестирование производительности (бенчмаркинг) таких приложений.

С другой стороны, в сообществе облачных технологий появились инструменты, упрощающие развёртывание и управление. Например, Kubernetes стал де-факто стандартом для декларативной оркестрации. Мы предлагаем использовать устоявшиеся облачные технологии вместе с паттерном Operator для создания инструментов бенчмаркинга, обеспечивающих воспроизводимость результатов.

После краткого введения в Kubernetes и паттерн Operator мы представляем модель данных и архитектуру системы для оператора бенчмаркинга.

**2. Паттерн Kubernetes Operator**  
Kubernetes — это декларативный инструмент оркестрации для облачно-ориентированных приложений. Пользователи описывают желаемое состояние приложения в виде ресурсов (resources), а Kubernetes в непрерывном цикле (reconciliation loop) изменяет текущее состояние системы для достижения желаемого. Типичный пример ресурса — Deployment, который описывает компонент приложения: образы контейнеров, количество реплик и другие параметры. Пользователи обычно взаимодействуют с Kubernetes через YAML-файлы, применяя их с помощью CLI-инструмента kubectl.

Паттерн Kubernetes Operator — это современный подход для интеграции предметно-ориентированных знаний в процесс оркестрации. Его реализация включает два этапа:

1. **Определения пользовательских ресурсов (Custom Resource Definitions, CRDs)** — создание новых типов ресурсов, управляемых через API Kubernetes.
2. **Оператор** — специализированный программный компонент, работающий внутри кластера Kubernetes и управляющий жизненным циклом объектов, описанных в CRDs. Оператор взаимодействует с API Kubernetes и реагирует на создание, изменение или удаление пользовательских ресурсов (Custom Resources, CRs — экземпляров CRDs).

**3. Пользовательские ресурсы для бенчмаркинга**  
Паттерн Operator объединяет знания об управлении Kubernetes с предметной областью. В контексте бенчмаркинга мы предлагаем описывать операционные знания в CRDs. В процессе тестирования можно выделить двух участников:

* **Разработчик тестов (Benchmark Designer)** — обладает знаниями о работе тестируемой системы (System Under Test, SUT) и методах измерения целевых показателей.
* **Исполнитель тестов (Benchmarker)** — запускает готовые тесты.

На основе этого разделения ролей мы предлагаем модель данных, где тесты и их выполнения разделены. Для каждого из них создаются отдельные CRDs:

* **Тесты (Benchmarks)** определяются разработчиками. Они описывают SUT, генерацию нагрузки и параметры метода тестирования. Тесты могут версионироваться и публиковаться в репозиториях.
* **Выполнения (Executions)** описывают отдельный запуск теста. Они ссылаются на конкретный Benchmark и настраивают экспериментальное окружение. При развёртывании в Kubernetes Executions получают состояния: Pending (ожидание), Running (выполнение), Finished (завершено) или Failed (ошибка).

**4. Архитектура оператора бенчмаркинга**  
Стандартные компоненты для бенчмаркинга включают:

* Тестируемую систему (SUT).
* Генератор нагрузки.
* Контроллер экспериментов.
* Базу данных измерений.
* (Опционально) Мониторинг для пассивного наблюдения.

Мы предлагаем архитектуру инструмента бенчмаркинга на основе оператора Kubernetes.

Для запуска тестов Benchmarker регистрирует тесты и инициирует их выполнение через CRs. Оператор постоянно отслеживает изменения в списке Executions через API Kubernetes. Он действует как контроллер экспериментов и решает (на основе политик инструмента), запустить выполнение немедленно или добавить его в очередь.

После старта Execution оператор создаёт SUT и генератор нагрузки через API Kubernetes, а также настраивает кластер. Во время выполнения оператор обновляет состояние Execution (например, на Running).

Состояния и события Execution управляются через API Kubernetes и используются для обратной связи с **исполнителем тестов**. В течение всего теста Prometheus собирает метрики мониторинга от SUT и генератора нагрузки. После завершения оператор останавливает и удаляет SUT и генератор, запрашивает данные из Prometheus и анализирует их. Возможен также анализ данных во время выполнения для адаптации SUT и нагрузки.

Результаты тестов сохраняются оператором в постоянное хранилище, откуда исполнитель тестов может их получить для анализа, архивации или обмена.

**5. Обсуждение и выводы**  
Паттерн Kubernetes Operator позволяет управлять конфигурацией приложений на уровне оркестрации. Применение этого подхода к бенчмаркингу облачно-ориентированных приложений даёт следующие преимущества:

* Чёткое разделение инструмента тестирования, теста и его выполнения.
* Тесты определяются в строгом формате и публикуются как описательные файлы без привязки к инструментарию.
* Вся экспериментальная настройка объединена в одном файле Execution.
* Управление тестами и выполнениями через стандартные инструменты Kubernetes (например, kubectl), что улучшает пользовательский опыт без дополнительной разработки.

В целом, использование паттерна Operator для создания инструментов бенчмаркинга повышает повторяемость и, как следствие, воспроизводимость исследований облачно-ориентированных систем.