Vrije Universiteit Amsterdam

Master Thesis

Testing in Kubernetes: A Use Case Study of JetBrains CodeCanvas

---

Author: He Wen (2813466)

1st supervisor:	Daniele Bonetta	
daily supervisor:	Matthijs Jansen	
daily supervisor:	Alexander Chumakin	(JetBrains)
2nd reader:	Tiziano De Matteis	

A thesis submitted in fulfillment of the requirements for the VU Master of Science degree in Computer Science

August 18, 2025

Abstract

As digital services grow in size and complexity, many systems adopt containerization as a lightweight, distributed solution. Kubernetes, one of the most popular container orchestration platforms, is widely used not only to manage containerized services but also to implement custom application-specific control logic on top of the Kubernetes control plane. Systems that embed such custom logic within Kubernetes are referred to as Kubernetes-integrated systems, where the application behavior depends on both the deployed containers and the underlying custom orchestrator.

However, testing such systems presents unique challenges. Existing research primarily addresses either Kubernetes-native components or application functionality deployed on Kubernetes, leaving a gap in end-to-end frameworks capable of validating both custom orchestration logic and core Kubernetes components.

This thesis addresses this gap using JetBrains CodeCanvas, a remote development platform that orchestrates containerized environments, as a case study. We propose a methodology for orchestration-aware test framework design and implement it as an end-to-end testing framework that (i) captures interactions between custom Kubernetes logic and core cluster components, (ii) supports validation across cluster upgrades to simulate real-world updates, and (iii) integrates with the CI/CD pipeline for continuous execution. A tailored test suite for CodeCanvas is designed and implemented for three core use cases: (i) component correctness, (ii) data integrity, and (iii) cross-version component compatibility. A thorough evaluation of the framework and test suite is performed in production environments, demonstrating that the framework detects critical bugs that would remain undetected without orchestration-aware testing mechanism.

Contents

1

Introduction

As digital services become more and more central to modern life, the infrastructure that supports them has expanded significantly in both complexity and scale. From banking and healthcare to communication and entertainment, more aspects of society rely on dependable, high-performing software systems. As digital services scale rapidly, **containerization** has become a fundamental technology within service infrastructure. Containers package applications and their dependencies into lightweight, portable units that can run consistently across different computing environments. Unlike traditional virtual machines (VMs), containers are faster to start, require fewer system resources, and are easier to manage (1). They allow companies to serve millions of users across regions while minimizing operational costs and infrastructure management overhead.

However, as systems grow in size and complexity, managing thousands of containers across clusters of machines becomes increasingly difficult. This has led to the rise of **container orchestration systems**, which helps automate the deployment, scaling, and maintenance of containerized applications. These tools provide a way to manage containers as a unified system, handling tasks such as service discovery, load balancing, and fault recovery. By simplifying both initial deployments and ongoing management of multiple containers as a single unit, orchestration systems have become a key part of modern software service infrastructure.

Among these systems, **Kubernetes** (2) has emerged as the most widely adopted solution. It provides a powerful platform for orchestrating containerized workloads as it manages application lifecycles across clusters, ensuring that services remain available, responsive, and scalable. According to the Cloud Native Computing Foundation Annual Survey (2024), approximately 93% of organizations are currently using or evaluating Kubernetes, reflecting its widespread adoption across industries (3).

Beyond its role as a general orchestration platform, Kubernetes offers a programmable control plane that enables applications to integrate directly with its services through its API. Large-scale

systems such as JupyterHub and OpenShift, which are deployed on top of Kubernetes, often extend Kubernetes with custom controller components or resources, allowing application-specific orchestration logic to run alongside the native Kubernetes scheduling and lifecycle management mechanisms. Such **Kubernetes-integrated systems** leverage Kubernetes not just as a hosting environment but as an integral part of their functionality, coordinating components and system states through Kubernetes-native events and workflows. *JetBrains CodeCanvas*, a remote development environment orchestrator system studied in this thesis, exemplifies this category by embedding its orchestration logic directly within the Kubernetes control plane to manage development environments at scale.

1.1 Context

Given its central role in modern infrastructure, the correctness and reliability of Kubernetes directly influence the stability of the systems built on top of it. Failures in orchestration logic can lead to service disruptions, security vulnerabilities, and broader system failures. This makes thorough testing not only essential for Kubernetes itself but also for Kubernetes-integrated systems, where Kubernetes is embedded as a critical operational layer to ensure workload availability, performance, and fault tolerance.

Testing Kubernetes itself presents unique challenges. Traditional testing methodologies often fall short in capturing the complexity of Kubernetes due to its dynamic and distributed nature. Containers can be created, terminated, rescheduled, or moved across nodes in real-time, making it difficult to maintain consistent testing conditions or predict system behavior under all possible scenarios (4, 5).

Moreover, the complexity increases further when testing Kubernetes-integrated systems. Testing these systems involves not just application logic but also configuration, orchestration behavior, networking, and resource management. Testing for scalability and resource management requires specialized techniques that account for dynamic resource allocation (6). Security testing is similarly challenging due to the risks of container and network isolation (7, 8). Additionally, failure modes are varied, with failures and partial outages that traditional tests might overlook (9, 10).

To address these pain points and challenges, there has been **active and continuing research** in Kubernetes-related testing, including functional testing techniques for validating application or native Kubernetes orchestration logic (11, 12), fault injection and tolerance frameworks to assess system resilience (13, 14), performance benchmarking of Kubernetes under varying load conditions (15, 16), and tools for security validation across the container orchestration lifecycle (17, 18, 19, 20).

However, many of these approaches remain **narrowly focused**. They often focus on isolated aspects and assume idealized environments, such as testing either single Kubernetes components or application-level behavior only. This leaves a critical gap for **end-to-end testing** that evaluates the entire system under realistic conditions, including both the Kubernetes layer and the application layer. Such testing is essential for Kubernetes-integrated systems, where native Kubernetes orchestration logic, custom Kubernetes controls, and application-level functionality must work together reliably. As a large-scale application with deep integration with Kubernetes, *CodeCanvas* serves as a great use case for researching methodologies and frameworks for testing large-scale systems that are not only deployed in Kubernetes but also implement application-specific orchestration logic with custom Kubernetes components.

1.2 Problem Statement

For Kubernetes-integrated systems like *CodeCanvas*, their deep coupling with Kubernetes introduces a range of testing challenges that go beyond the scope of existing methodologies. *There currently exists no systematic methodology or automated end-to-end framework for testing systems that are simultaneously applications built on Kubernetes and infrastructure tools deeply integrated with its control plane.*

Testing such systems demands orchestration-aware testing strategies that can validate not only application behavior but also underlying Kubernetes components under real-world deployment conditions at the end-to-end level. In the context of a large-scale system like *CodeCanvas*, which is undergoing rapid development and for which new versions are continuously deployed, testing challenges become especially critical. One common scenario is *data loss following a version update*, where user workspaces or critical system metadata fail to persist across Kubernetes clusters or application upgrades. Another is *component version incompatibility*, where certain application or Kubernetes components are incompatible with existing dependencies after an update deployment, triggering failures across the system.

This thesis addresses the specific and timely problem of **designing an orchestration-aware testing framework as well as a testing suite** tailored to *CodeCanvas*, with the broader goal of establishing reproducible, automated, and comprehensive testing practices for a new class of infrastructure-integrated development platforms that operate natively within Kubernetes ecosystems.

1.3 Research Questions

Based on the problem statement, the main research question will be broken down to 3 research questions as follows:

- **RQ1: How can we design a continuous testing framework for Kubernetes-integrated systems?**
  Existing testing frameworks do not sufficiently address the need to verify both application correctness and the behavior of Kubernetes components manipulated by the application. Furthermore, for systems that are developed in an agile environment with continuous delivery, it is essential to maintain ongoing validation of correctness and compatibility after each update, while ensuring the flexibility and maintainability of the test infrastructure. In particular, for *CodeCanvas*, this research question focuses on designing a testing framework capable of validating critical application components, their interaction with Kubernetes, and the correctness of Kubernetes cluster configurations and updates.

- **RQ2: How can a effective test suite be constructed for Kubernetes-integrated systems?**
  A key aspect of effective testing is the identification of appropriate test subjects and the definition of their corresponding testing objectives. This research question aims to establish a systematic approach to selecting test subjects and designing relevant test scenarios.

  - **RQ2.1: Which components of a Kubernetes-integrated system must be tested?**
    This sub-question focuses on identifying the core application modules and the Kubernetes control plane components that are critical to system functionality. For *CodeCanvas*, particular emphasis is placed on the components responsible for environment provisioning, orchestration, and state management.
  - **RQ2.2: What are the expected behaviors and outcomes for different Kubernetes-based system components under various conditions?**
    This sub-question aims to define the expected operational behaviors, failure tolerances, and recovery mechanisms for the components identified in **RQ2.1**, under realistic orchestration events such as cluster upgrades and resource contention.

- **RQ3: How can the effectiveness of the designed testing framework and test suite be evaluated?**
  This research question addresses the development of evaluation criteria and metrics to assess the framework's effectiveness in terms of coverage, fault detection, reproducibility, and

integration with continuous delivery pipelines. Within the context of *CodeCanvas*, the evaluation is to ensure that the framework and the test suite can effectively verify the system's correctness across a range of real-world deployment scenarios and operational disruptions.

1.4 Research Methodology

This thesis utilizes a combination of design-based, use case-driven, and experimental research methods in order to systematically address the research questions:

- **M1: Design, abstraction, prototyping.**
  To answer **RQ1**, a continuous, orchestration-aware testing framework will be designed based on the functional and non-functional requirements of Kubernetes-integrated systems, with *CodeCanvas* serving as the primary reference system. The framework will be abstracted into a high-level model to generalize beyond a single application while ensuring applicability in continuous delivery contexts. A prototype implementation on *CodeCanvas* will be developed to validate the design decisions.

- **M2: Quantitative research.**
  To support **RQ2**, a systematic analysis will be conducted of the functionalities and interactions within Kubernetes and *CodeCanvas*. This includes surveying system responsibilities, identifying critical components, and formulating corresponding testing objectives. This structured analysis informs the design of a comprehensive and targeted test suite.

- **M3: Use-case study.**
  A detailed use-case study of *CodeCanvas* will be performed in this thesis. To answer **RQ2.1**, a detailed system study of *CodeCanvas* will be conducted. This will involve analyzing architectural documentation, reviewing operational traces, and collaborating with domain experts when available. To answer **RQ2.2**, operational scenarios and failure conditions will be systematically derived from *CodeCanvas*' expected behaviors.

- **M4: Experimental research, quantitative evaluation.**
  To answer **RQ3**, the designed testing framework and test suite will be deployed and evaluated against *CodeCanvas* under diverse real-world operational conditions. Effectiveness will be measured using quantitative metrics including test coverage and fault detection rate.

1.5 Thesis Contributions

This thesis makes the following contributions:

- **C1 (Conceptual): Design and implementation of a continuous, orchestration-aware testing framework for Kubernetes-integrated systems.**
  A testing framework is proposed that specifically targets systems operating both as applications on Kubernetes and as infrastructure tools integrated with Kubernetes control-plane operations. The framework addresses gaps in existing testing methodologies by incorporating real-world orchestration dynamics and continuous integration/deployment requirements. **(RQ1)**

- **C2 (Conceptual): A methodology in end-to-end testing framework and suite design for Kubernetes-integrated systems.**
  A structured model is developed that identifies and categorizes the essential system and Kubernetes components that must be tested to ensure system correctness, based on a detailed case study of *CodeCanvas*. **(RQ2.1)**

- **C3 (Conceptual): Design and implementation of a systematic test suite for Kubernetes-integrated systems.**
  A comprehensive test suite is designed to define expected behaviors, failure conditions, and validation objectives for each critical component identified. The suite is built to cover a range of realistic orchestration events and simulation of real-word cluster updates, ensuring meaningful verification of complex Kubernetes-based systems. **(RQ2)**

- **C4 (Experimental): Quantitative evaluation of the testing framework and test suite.**
  The effectiveness of the designs is evaluated using key metrics such as test coverage, fault detection, reproducibility, and performance overhead in CI/CD pipelines. **(RQ3)**

1.6 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

1.7 Thesis Structure

This thesis is structured as follows: Chapter 2 introduces related background information, including the Kubernetes and CodeCanvas structures. Chapter 3 and Chapter 4 demonstrate the design

methodologies and results of the testing framework **(RQ1)** and the test suite **(RQ2)** for CodeCanvas. Chapter 5 illustrates the implementation of the designs. Chapter 6 answers **RQ3** by illustrating an evaluation experiment on the framework and suite implementation, including the experiment setup and results for different use case scenarios. Chapter 7 covers the current state of research on various topics related to our research questions. Chapter 8 concludes the thesis with answers to the research questions and suggestions for future work.

2

Background

2.1 Kubernetes

Kubernetes is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes has become a foundational technology for modern distributed systems. Understanding its core architecture is essential for identifying the testing objectives of Kubernetes-integrated systems. This section introduces the key architectural components of Kubernetes relevant to the context of systems integrated with Kubernetes logic.

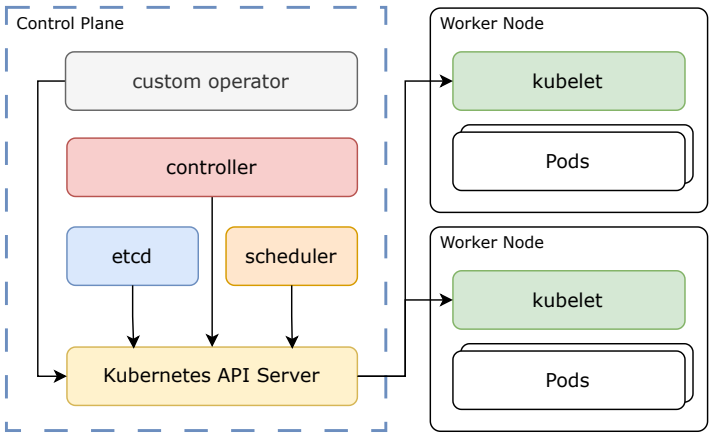


Figure 2.1: Architecture of Kubernetes

A Kubernetes cluster is composed of a control plane and one or more worker nodes. The control plane is responsible for maintaining the desired state of the system, while worker nodes are responsible for running the actual application workloads.

The **worker node** is the primary execution environment for user workloads. Each node runs containers in the *pods* at runtime. Meanwhile, *kubelet* is responsible for communicating with the control plane.

The **control plane** components govern cluster-wide decisions and maintain the cluster state. The *Kubernetes API server* is the central access point for all operations within the cluster. It exposes a RESTful interface through which components interact with the Kubernetes system. *Etcd* is a distributed key-value store that serves as the persistent backing storage for all cluster states. The *scheduler* assigns unscheduled pods to the appropriate worker nodes. The *controller* continuously monitors the cluster state and attempts to transition it to the desired state.

One of the defining features of Kubernetes is its declarative control model, which enables extensibility through custom logic embedded in the control plane without modifying the code of Kubernetes itself. A *custom operator* is a higher-level controller built using custom resources to manage applications and components in Kubernetes. It automates complex operational tasks, such as installation, configuration, and scaling, by interacting with *the Kubernetes API server*. Many Kubernetes-integrated systems, including OpenShift and JupyterHub, use *custom operators* to implement application-specific orchestration logic.

2.2 JetBrains CodeCanvas

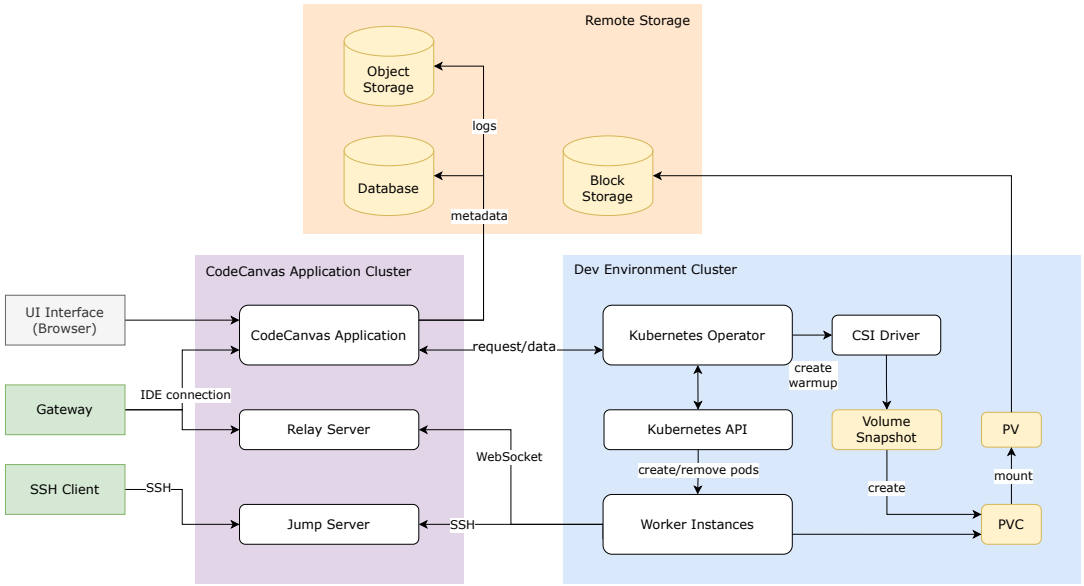


Figure 2.2: Architecture of CodeCanvas

CodeCanvas is a cloud-based development environment platform developed by JetBrains. It functions as a remote development orchestrator by provisioning and managing containerized development environments in the cloud, thereby minimizing infrastructure overhead and reducing the time required for configuring local environments.

The setup process begins with the definition of an *environment configuration template*, which specifies parameters such as source code repositories, integrated development environment (IDE) versions, and other tooling dependencies. From a single template, multiple isolated *development environments* can be created. These environments connect to a user's local IDE interface, while all computation and data storage remain within the cloud-hosted CodeCanvas infrastructure. Moreover, users may generate a *warm-up snapshot* from a template. The snapshot will preload dependencies and project metadata, accelerating the provisioning of new development environments.

Figure 2.2 illustrates the underlying architecture of CodeCanvas. CodeCanvas consists of several Kubernetes clusters, one **CodeCanvas application cluster**, and one to multiple **Development environment clusters**. **CodeCanvas application cluster** hosts the *CodeCanvas Application*, the main back-end service component, as well as the *Relay Server* and *Jump Server*. The **development environment cluster** is responsible for hosting development environment workers and integrating deeply with the Kubernetes architecture.

CodeCanvas Application. This component serves as a user-facing web service that provides the interface for managing development environments through browsers. It also connects to a Kubernetes Operator using the compute API, provisioning resources indirectly.

Relay Server and Jump Server. These two intermediary components serve as a secure connection between the user machine and the development environment instances. *Relay Server* provides WebSocket connections between IDE services on the user machine and the development environments. Similarly, *Jump Server* forwards SSH connections between the SSH client and the development environments.

Kubernetes Operator. This component is a customized Kubernetes controller that embeds the domain-specific logic of CodeCanvas to deploy, configure, and manage the lifecycle of worker instances by manipulating the native Kubernetes RESTful API.

Worker Instances. A worker is a Kubernetes-managed, specialized runtime agent responsible for controlling the lifecycle of development environments and warm-up tasks. Each development environment is deployed as a Kubernetes Pod that contains a single worker container. Each worker

instance is implemented as a container running inside a Kubernetes pod and orchestrates environment bootstrapping, runtime coordination, and state reporting.

External Storage. The external storage, including the *database*, *object storage*, and *block storage*, is stored outside the clusters for persistent data. **CodeCanvas Application** invokes the *database*, which stores the application's state, development environment states, user data, service accounts, personal secrets, and other metadata. The *object storage* is responsible for providing development environment logs and audit log data for the *CodeCanvas Application*. The *block storage* stores persistent volumes of user data and code for development environments.

3

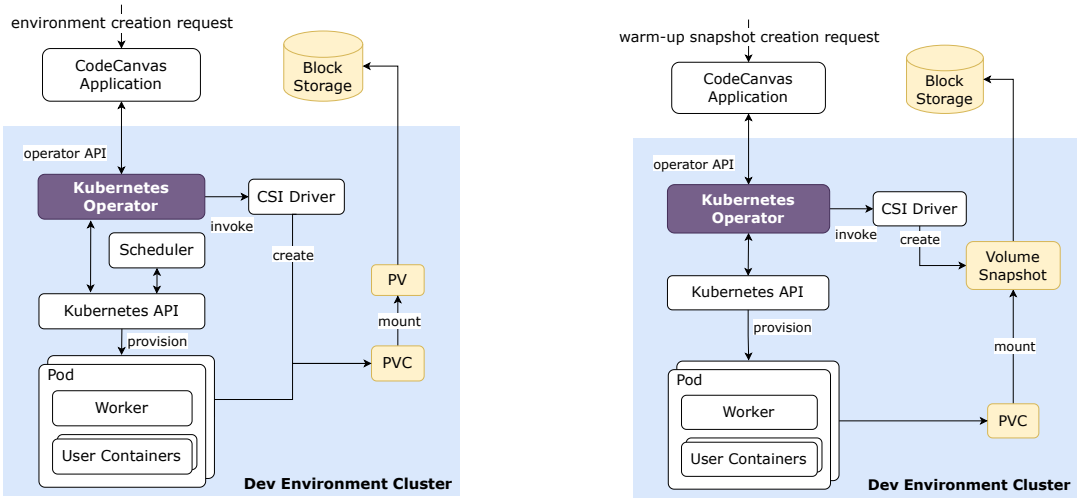
Design of the Testing Framework

In this section, to address ***RQ1:*** *How can we design a continuous testing framework for Kubernetes-integrated systems*, we conduct a requirements analysis, including the analysis of the CodeCanvas workflow integrating with Kubernetes (Section 3.1), and the identification of functional (Section 3.2.1) and non-functional (Section 3.2.2) requirements of the testing framework. Then, we introduce the framework architecture by presenting an overview of the workflow and the responsibilities of each framework component, while emphasizing the orchestration-aware mechanism of the framework (Section 3.3).

3.1 System Workflow Analysis

CodeCanvas is a cloud-based remote development solution that orchestrates containerized development environments. Users can create environments from predefined repository and IDE configuration templates or speed up environment creation through pre-configured warm-up snapshots. As a Kubernetes-integrated system, *CodeCanvas* not only deploys within a Kubernetes cluster but also interfaces directly with native Kubernetes APIs and storage mechanisms to implement custom orchestration logic. At the core of this functionality is a customized Kubernetes operator, which is a Kubernetes extension mechanism that enables the definition of customized control logic. This operator processes requests from *CodeCanvas* application components and coordinates corresponding state changes within the Kubernetes cluster.

Therefore, designing an end-to-end testing framework for *CodeCanvas* requires a comprehensive understanding of the interactions between three layers: the application components, the custom Kubernetes operator, and the native Kubernetes subsystems. A detailed system workflow analysis enables the identification of critical integration points and interaction patterns across



(a) Development environment creation

(b) Warm-up snapshot creation

Figure 3.1: CodeCanvas application main workflows

these layers. This analysis forms the basis for defining both functional and non-functional testing requirements and provides architectural guidance for building a testing framework capable of validating the system in realistic operational conditions.

Two main workflows of CodeCanvas are analyzed in the following sections, covering the development environment initialization process and the storage management of CodeCanvas using Kubernetes components.

Development Environment Lifecycle. A development environment is a containerized workspace for end users, including code repositories, IDE backend, and required development tools. As shown in Figure 3.1a, each development environment is deployed as a Kubernetes Pod that contains a single worker container and nested user containers. When a development environment creation request is made to the **CodeCanvas Application**, it forwards the request and configuration data to the **Kubernetes Operator**. Then the **Kubernetes Operator** creates the respective task, enqueues the task, and delegates it to the Kubernetes scheduler by deploying a new pod. If sufficient cluster resources are available, the pod is scheduled and initialized. Otherwise, the task remains in a pending state until resources are freed or additional compute capacity is provisioned. Once the new pod is created, it then initializes the development environment by orchestrating the creation of nested containers, including the worker container and multiple user containers hosting user data and microservices. Once the development environment is terminated by the user

or the platform, the worker shuts down all nested containers and exits. The Kubernetes Pod is then deleted by the **Kubernetes Operator**, ensuring the complete statelessness of the pod while maintaining stateful storage via the retained volume.

Apart from managing and monitoring user container lifecycles, the worker is also responsible for managing the persistent storage associated with each development environment. Prior to launching the development environment pod, the **Kubernetes Operator** provisions a Persistent Volume Claim (PVC), which is dynamically bound to a Persistent Volume (PV) in external **Block Storage** through the Container Storage Interface (CSI) driver in Kubernetes. This volume is then mounted into the development environment pod and used to persist user data, such as source code, configurations, and environment state. When a development environment is stopped, the volume is safely unmounted and preserved. It is later reattached if the environment is restarted, thus ensuring data persistence across sessions.

Warm-up Snapshots. To optimize the startup latency of development environments, CodeCanvas employs a warm-up mechanism that leverages Kubernetes-native storage abstractions in conjunction with the CSI snapshot functionality. The warm-up process is designed to pre-execute environment initialization tasks and persist the resulting system state as a reusable snapshot. This snapshot serves as a baseline image from which future development environments can be instantiated rapidly and with minimal overhead.

As shown in Figure 3.1b, a warm-up task is conceptually similar to a standard development environment deployment, which is encapsulated within a Kubernetes Pod running a worker container and nested user containers. During warm-up execution, the worker performs all predefined bootstrapping tasks and writes the resulting environment state to an associated PV, dynamically provisioned via a PVC. Upon successful completion of the warm-up task, the **CSI Driver** creates a **Volume Snapshot** object. This object references a specific PVC and serves as a metadata descriptor for a point-in-time snapshot of the volume. The snapshot is then stored in the block storage.

When a user requests the creation of a new environment, the **Kubernetes Operator** specifies a field in the PVC that references the existing **Volume Snapshot**. The CSI driver then provisions a new volume initialized with the snapshot's contents, thereby bypassing time-intensive setup steps.

Results. In the context of testing orchestration-aware Kubernetes-based systems, the workflows of CodeCanvas provide insights on design requirements and architectural design decisions of the testing framework:

- **Insight 1: Interactions between the application and Kubernetes control plane.** The tight coupling of the application and Kubernetes means that any system change must be evaluated in terms of both application logic and its Kubernetes orchestration behavior.
- **Insight 2: Versioned state artifacts.** The snapshot-based warm-up feature introduces versioned state artifacts that may persist across multiple releases, thus posing a challenge in ensuring compatibility between new application logic and old data artifacts.

3.2 Requirements Analysis

With the goal of building an end-to-end testing framework for Kubernetes-integrated systems, we analyze the CodeCanvas use case, and based on the insights, derive its functional and non-functional design requirements.

3.2.1 Functional Requirements

The functional requirements (FRs) define the expected crucial behaviors and functions that the testing framework should have.

- **FR1: Multi-layer coverage.** CodeCanvas is an orchestrator that manipulates Kubernetes-native resources via a customized operator. Therefore, the testing framework must account for both layers simultaneously, validating not only application correctness but also correctness in orchestration patterns and cluster-state transitions. **(Insight1)**
- **FR2: Cluster lifecycle management.** The framework must support provisioning, snapshotting, and restoring Kubernetes clusters with specific versions of the target system in order to test versioned state artifacts and system compatibility after applying system updates.
- **FR3: Test data injection and state preparation.** The framework must enable creation and injection of realistic test data, such as development environments, user accesses, cloud policies and warm-up snapshots, so that it can validate the integrity of data artifacts after the simulated system update.
- **FR4: Custom test case execution.** The framework must support running customized test cases, including functional tests, integration tests, and regression tests. Since the testing scenarios of CodeCanvas vary across system configurations, supporting a wide spectrum of custom test logic ensures the framework adapts to evolving requirements and different use cases.

- **FR5: Result validation and reporting.** The framework must provide mechanisms for validating data integrity, system responsiveness and correctness, and Kubernetes resource stability. Meanwhile it must provide readable test execution reports for test framework users so that orchestration or application failures can be quickly diagnosed without intensive manual log analysis.
- **FR6: Integration with CI/CD pipelines.** The framework must be invocable on CI platforms as CodeCanvas is a system with rapid iteration cycles. Meanwhile, a framework developed in the context of CI/CD has generalization for large Kubernetes-based systems with frequent updates similar to CodeCanvas.

3.2.2 Non-Functional Requirements

The non-functional requirements (NFRs) define the quality attributes of the testing framework.

- **NFR1: Scalability.** The framework must handle scaling to test large clusters or simulate many parallel development environments, as CodeCanvas deployments may involve parallel setups or a high number of concurrent sessions in production.
- **NFR2: Fault isolation.** Test runs must be isolated and recoverable from partial failures. Failures in one test run must not affect other test runs. This prevents false positives and ensures that one faulty scenario does not compromise unrelated results.
- **NFR3: Extensibility.** It must be easy to extend the framework with new types of test scenarios and test cases, additional Kubernetes events, and new validation modules or assertions since CodeCanvas and Kubernetes are constantly evolving.
- **NFR4: Security and access control.** The framework should respect access boundaries and security permissions. It must protect sensitive user data and prevent the exposure of infrastructure-level vulnerabilities.

3.3 Framework Architecture

Using the design methodology for frameworks proposed by Wieringa et al. (21), we translate the functional and non-functional requirements into the proposed testing framework, which is designed to support the continuous integration and compatibility verification of Kubernetes-integrated systems such as CodeCanvas. It is structured into five core components, addressing the functional and non-functional requirements. The main components include the **test controller**,

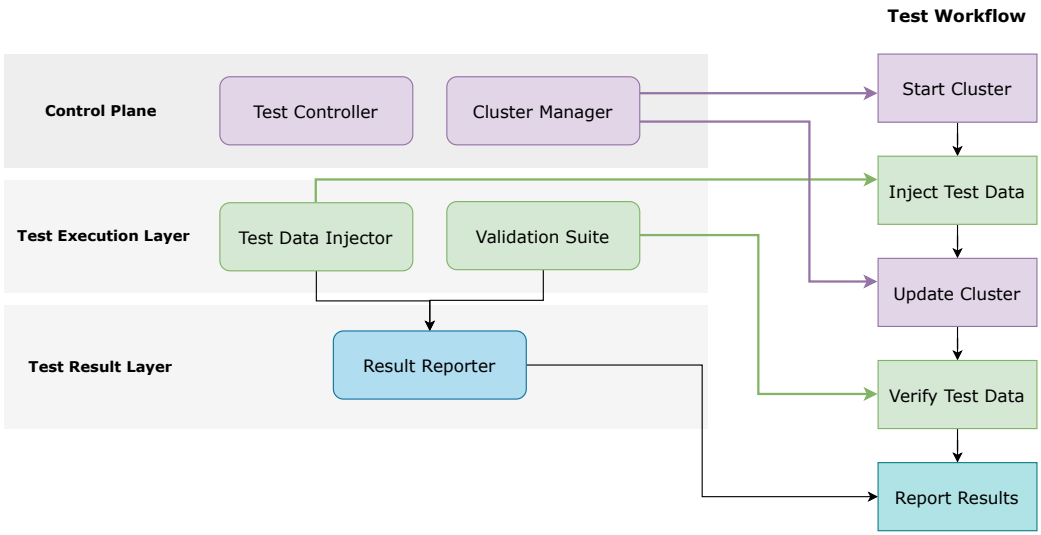


Figure 3.2: Mapping between testing framework components and test workflow

cluster manager, **test data injector**, **validation suite**, and **result reporter**. This section presents an architectural overview of the framework, including its layered design, component responsibilities, high-level workflow, and the assumptions about the underlying system behavior and test environment. In particular, we highlight how the framework achieves orchestration awareness by interacting with both application-level and Kubernetes-native components.

3.3.1 Design Overview

Existing Kubernetes testing tools often focus on unit tests at the pod or deployment level [cite]. In CodeCanvas, meaningful correctness guarantees depend on interactions between components across cluster boundaries, such as the coordination between the Kubernetes Operator, worker containers, and CSI-managed storage. Therefore, we present a testing framework for CodeCanvas that validates cross-cluster correctness, as shown in **Figure 3.2**.

The components of the testing framework are divided into three layers of functionality: control plane (**test controller**, **cluster manager**), test execution layer (**test data injector**, **validation suite**), and test result layer (**result reporter**). The control plane is responsible for the test life cycle, controlling the test environment, and coordination of the test execution and result report layer. The test execution layer provides the test data preparation and validation by interacting with the CodeCanvas application and Kubernetes native components (**FR1**). The test result is gathered, processed, and reported by the test result layer. The detailed responsibility of each core component is as follows:

- **Test Controller.** The test controller is the central orchestrator of the testing framework. It is responsible for orchestrating the full life cycle of test execution, including test setup, cluster upgrade, data validation, and test result reporting. It interfaces directly with the CI/CD pipeline (**FR6**), ensuring automated triggering and integration of the testing process.

- **Cluster Manager.** The cluster manager is responsible for provisioning, updating, and tearing down Kubernetes clusters used in testing, including both CodeCanvas application clusters and development environment clusters. It also handles version control of the candidate CodeCanvas application according to **FR2**, as well as infrastructure deployment, such as Kubernetes operator, jump server, and relay server.

- **Test Data Injector.** The test data injector is responsible for the preparation of test data, including development environments, user data, stateful volumes, and snapshots, implementing **FR3**. It supports the creation of custom test data aligned with specific validation objectives, ensuring stateful workloads are accurately simulated and testable across upgrades.

- **Validation Suite.** The validation suite encapsulates the test oracles and executes system-level and component-level test cases. It interacts with the CodeCanvas application and Kubernetes-native components to verify system behavior under various operational scenarios, thereby addressing **FR4**.

- **Result Reporter.** The result reporter collects runtime logs, system metrics, and validation outcomes from the **Validation Suite**. It generates user-readable reports and artifacts that are compatible with CI tooling, satisfying **FR5**. This ensures test feedback is easily interpretable and traceable within development workflows.

The overall workflow of the testing framework begins with the configuration and triggering phase, where the **test controller** receives the definition of test inputs and associated validation logic. Once triggered, the controller invokes the **cluster manager** to determine a reference benchmark version of the system and provision the corresponding Kubernetes clusters, which include both the CodeCanvas application and development environment clusters. After the benchmark clusters are initialized, the **test controller** activates the **test data injector** to populate the system with predefined stateful workloads, such as development environments, persistent volumes, and warm-up snapshots. Upon successful data injection, the **cluster manager** proceeds to perform a system upgrade, deploying the candidate version of the application under test to simulate an actual version rollout scenario, updating all the CodeCanvas-related components (**CodeCanvas Application, Relay Server, Jump Server, and Kubernetes Opeator**). The **validation suite** is then

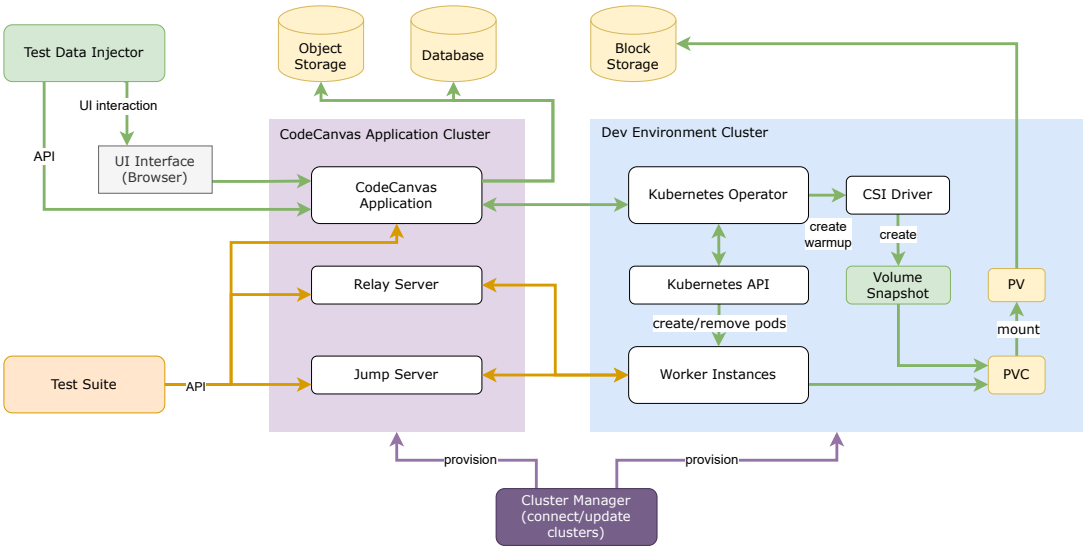


Figure 3.3: Orchestration-awareness mechanism of the testing framework

executed to run test cases that evaluate application-level correctness and Kubernetes-level integration. Finally, the **result reporter** collects runtime logs, system metrics, and validation outcomes, generating structured reports and artifacts. This end-to-end workflow ensures the correctness of system functionality, user data integrity, and orchestration features while maintaining alignment with continuous integration and delivery pipelines.

3.3.2 Orchestration-Aware Mechanism

As mentioned in [background], the test framework is designed to run across two main Kubernetes clusters: the application cluster, which hosts core platform services, and the development environment cluster, which manages user-facing containers and persistent workloads. A core feature of the framework is its orchestration-aware capabilities. It interacts not only with application-level interfaces, but also with Kubernetes-native components that underpin system behavior. This subsection outlines how the framework utilizes orchestration awareness through three key components.

- **Test Data Injector.** The test data injector prepares the system for validation by automating both user-facing and internal interactions. It populates the system state by emulating browser-based interactions with the CodeCanvas UI and invoking backend service APIs directly. For user data initialization, the injector communicates with the **CodeCanvas Application**, which in turn persists the data to the backing **database**. To set up development

environments and warm-up snapshots, the injector triggers environment creation via the application APIs; the **CodeCanvas Application** then forwards these operations to the **Kubernetes Operator**, which provisions the corresponding pods and attaches persistent volumes via native Kubernetes mechanisms.

- **Validation Suite.** The validation suite is responsible for executing end-to-end integration and regression tests across both layers of the system. It validates system correctness by invoking APIs on the **CodeCanvas Application** and establishing connections to the runtime worker instances via the **Relay Server** and **Jump Server**. These interactions allow the framework to verify key orchestration properties, including the integrity of development environment bootstrapping, the correct functioning of the **Kubernetes Operator**, and the persistence and consistency of user data stored in external volumes across deployment updates.

- **Cluster Manager.** The cluster manager connects the two main Kubernetes clusters in a control-plane level. It plays a critical role in enabling orchestration-aware testing by managing the lifecycle and configuration of Kubernetes clusters involved in test execution. It provisions both the **CodeCanvas application cluster** and the **development environment cluster**, ensuring that each test scenario is executed in a controlled and reproducible infrastructure context.

Together, these components ensure that the testing framework can validate not only the application logic but also the complex interactions between user workloads and Kubernetes orchestration, which are central to the reliability of infrastructure-integrated platforms like CodeCanvas.

3. DESIGN OF THE TESTING FRAMEWORK

4

Design of the Test Suite

This section addresses *RQ2: How can a comprehensive test suite be constructed for Kubernetes-integrated systems* by answering *RQ2.1* and *RQ2.2*, identifying core components to be tested and defining the expected outcomes and assertions of each component.

To systematically construct the test suite for CodeCanvas, we adopt three-step design methodology that begins with a **structural and behavioral analysis** of the system under test (Section 4.1). Based on this analysis, we define a set of **validation goals** that specify the key functionalities, interactions, and guarantees that must be verified (Section 4.2). We then introduce a set of **design principles** that guide how validation goals should be instantiated (Section 4.3). Finally, we translate these goals and principles into a concrete **test suite** architecture, including a generic test case model and a set of representative test scenarios that ensure comprehensive and maintainable validation coverage for CodeCanvas.

4.1 Component Analysis

To answer *RQ2.1: Which components of a Kubernetes-integrated system must be tested*, this section presents a comprehensive analysis of CodeCanvas components and their interactions with native Kubernetes, in order to identify each component's responsibilities, its failure modes, and the corresponding testing strategies necessary to ensure system correctness and robustness, thus informing the design of the test suite.

CodeCanvas Application. The CodeCanvas Application acts as the user-facing backend server that handles API requests related to development environment creation, warm-ups, storage management, and project configurations. It handles user administration and stores user-related metadata into the database. It also coordinates with the **Kubernetes Operator** and manages configu-

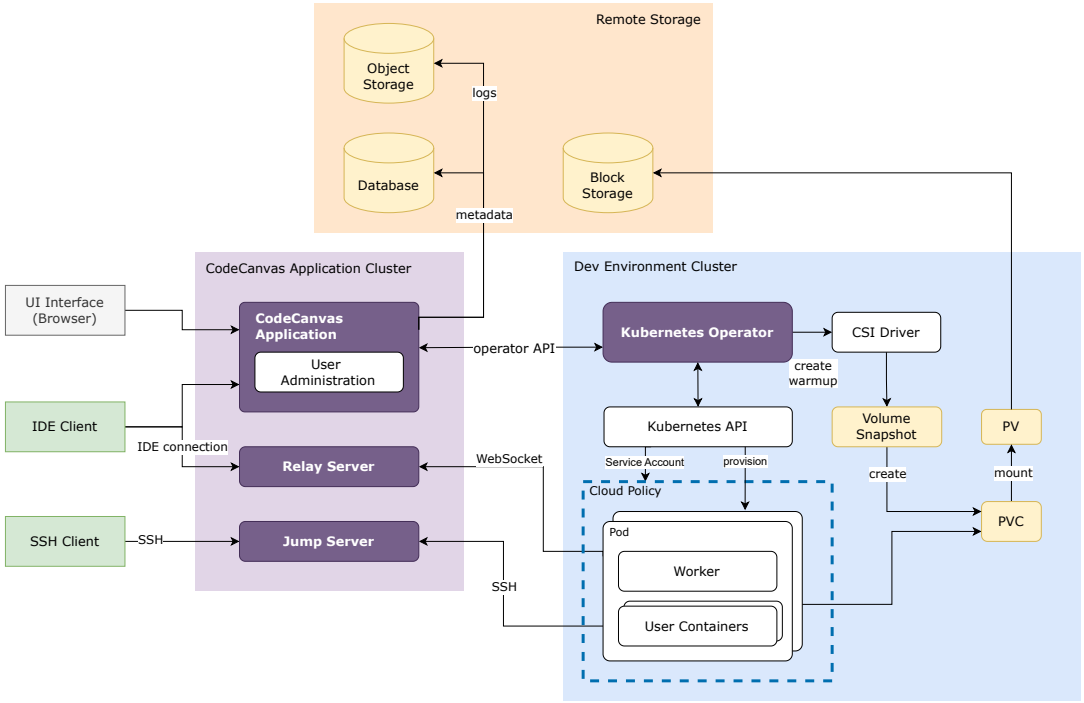


Figure 4.1: Core components of CodeCanvas

ration metadata and state via database and object storage. As the entry point for most operations, the CodeCanvas Application is critical to both functional correctness and cross-component coordination. Failures in this component can include backward-incompatible API changes, data persistence failures, or miscommunication with dependent components such as the **Kubernetes Operator**. The test suite must therefore validate API contract stability, enforce data integrity through end-to-end assertions, and ensure compatibility across releases.

Kubernetes Operator. The Kubernetes Operator is responsible for translating abstract environment requests into Kubernetes-native resources such as pods, PVCs, and VolumeSnapshots. It carries orchestration logic, managing complex lifecycle events including warm-up executions, nested container initialization, and storage provisioning. Errors in the operator can include request processing errors, inconsistent resource states, failed pod scheduling, unbound volumes, or invalid snapshot creation. The test suite must validate not only the final system outcomes but also intermediate transitions such as PVC binding, pod readiness, and CSI snapshot lifecycle.

Relay Server and Jump Server. The servers act as a network intermediary, forwarding terminal traffic between clients on user machines and remote development environments by managing

WebSocket and SSH connections. Failures in servers include broken or unauthenticated connections, version incompatibilities with upstream components, or relay misconfigurations. Therefore, the test suite should include runtime connectivity validation and testing across server versions to detect protocol or handshake mismatches.

Worker. Implemented using Kubernetes service accounts, cloud polices allow users to set up automatic authentication of development environment pods in cloud services. Within each development environment pod, the **Worker** is responsible for launching nested user containers, managing mounted volumes, and monitoring runtime execution. It operates under pod environments and must remain tightly coupled with the Kubernetes storage and scheduling subsystems. Failures at this level may involve improper volume mounting, incorrect container sequencing, bootstrapping errors, or resource exhaustion. The test suite should verify correct initialization and teardown behavior, validate the success of warm-up routines, and ensure data consistency across environment resume operations.

Persistent Storage. CodeCanvas also relies on Kubernetes-native persistent storage, primarily implemented through dynamically provisioned PVCs and Kubernetes internal **CSI Driver** orchestrated by the **Kubernetes Operator**. Potential failures of storage provisioning can include snapshot corruption, PVC rebinding failures during upgrades, or misaligned CSI configurations. The test suite must validate volume lifecycle operations, including snapshot creation and restoration, and data integrity across cluster upgrades.

4.2 Validation Goals

Understanding the functional responsibilities and possible failure modes of each core component within CodeCanvas provides insights into identifying what must be tested to ensure system correctness, robustness, and cross-version compatibility. A set of validation goals that the test suite must address across different scenarios is defined based on the component analysis.

Each validation goal (VG) targets a critical system behavior or component interaction that must be preserved.

- **VG1: Development environment lifecycle correctness.** CodeCanvas provisions development environments as orchestrated Kubernetes Pods that integrate worker and nested user containers. The test suite must validate the full lifecycle of pod provisioning, initialization,

runtime behavior, and teardown. This includes verifying coordination between the CodeCanvas application, the Kubernetes Operator, and native Kubernetes mechanisms to ensure environments are correctly managed in response to platform triggers.

- **VG2: Warm-up snapshot validity.** Warm-up snapshots serve as reusable, pre-initialized system states that reduce development environment startup latency. To preserve their effectiveness, the test suite must validate that snapshots are generated correctly, capture meaningful system state, and can be reliably restored. This includes checking the snapshot creation pipeline via the Kubernetes CSI interface and the integrity of the corresponding PVs.
- **VG3: Persistent data integrity across versions.** CodeCanvas stores user data, environment metadata, and snapshots in externally provisioned storage volumes. The test suite must ensure that the data remains consistent and accessible across environment restarts and system version upgrades. In particular, tests must verify PVC-to-PV binding correctness, snapshot restorability, and data integrity following cluster updates.
- **VG4: Cross-version compatibility of core components.** Since CodeCanvas follows a continuous integration model, the test suite must validate that newer cluster versions remain interoperable with older infrastructure deployments and vice versa. This includes ensuring that existing clusters with legacy servers can function with updated backend logic, and that new features do not disrupt compatibility with persistent state or native Kubernetes APIs.

4.3 Design Principle

The design of the test suite is informed by the structural and behavioral characteristics of CodeCanvas as a Kubernetes-based system. The suite is constructed to ensure functional correctness, data persistence, and backward compatibility during continuous integration and frequent system upgrades. The following design principles (DP) reflect the rationale behind the structure and content of the test suite, and how they align with CodeCanvas-specific constraints.

- **DP1: Declarative and modular test definitions.** To support maintainability and extensibility, all test scenarios and test cases are defined declaratively using structured test manifests. It enables complex validation specification without embedding low-level application or Kubernetes native logic. Moreover, each test case is structured into modular stages to enable reusability and composability across different test targets, such as setup, validation, and teardown of volumes, warm-ups, and servers, which can be reused and composed across different test scenarios.

Table 4.1: Test definition model

Test Metadata	Test identifiers, natural language descriptions, relevant execution constraints
Preparation	User-generated inputs, development environment templates, warm-up snapshot definitions
Execution	Execution steps, post-test conditions, and clean-up behavior
Test Oracle	Expected outcomes, properties and behaviors, assertions

- **DP2: Orchestration-aware validation.** CodeCanvas provisions and manages Kubernetes pods and nested containers via a customized Kubernetes operator. The lifecycle spans multiple abstraction layers from application logic to customized operator actions to Kubernetes control plane scheduling to storage provisioning. Due to the deep integration of CodeCanvas and Kubernetes, the test suite is designed to be orchestration-aware, validating the behavior of the entire pipeline from user input to system response, including all intermediate resource transitions. Test assertions are based on observed system states, including PVC binding and snapshot readiness, rather than timeouts or blocking assumptions.

- **DP3: Version-robust test logic.** To support continuous integration and frequent release cycles, the test suite must remain stable and effective across changes in CodeCanvas versions. The test logic must account for version-specific differences without requiring interruptive rewrites, including the use of conditional execution paths, parameterized test definitions, and assertions under certain feature flags.

4.4 Test Suite Structure

In this section, we introduce the structure of the test suite following the design principles (Section 4.3). The structure of the test suite is designed to systematically capture complex validation logic in a maintainable and extensible form. It includes how test cases are defined, executed, and integrated with the testing framework, employing a declarative and modular model (**DP1**) around a separation between **test data injection** and **validation execution**.

Each test case in the suite is defined through a declarative specification that captures both configuration inputs and expected behaviors. As shown in Table 4.1, the structure of a test case consists of the following four primary components.

Test Metadata. Each test is annotated with a unique identifier, a descriptive summary, and applicable execution constraints such as execution environment configuration and triggering conditions, which enable conditional test inclusion during test runs, facilitating version-robust test logic (**DP3**). When integrating with the testing framework, the **test controller** interprets metadata to manage test registration, apply environment filters, and schedule execution across appropriate clusters and configurations.

Preparation. This component specifies the required system state and user data prior to initiating the test logic, including user-generated inputs to be stored in the database and object storage, configuration for development environments, warm-up snapshot definitions and creations. Preparation is carried out through both application-level APIs and automated UI flows to ensure test coverage of user-facing and cluster internal orchestration logic (**DP2**). The **test data injector** of the framework translates application API calls and UI interactions to CodeCanvas provisioning pipeline, including the interaction between the application cluster, custom Kubernetes operator, and native Kubernetes control plane.

Execution. This component defines the sequence of actions used to perform end-to-end verifications, ensuring that persistent data, environment states, and components reach the expected state. It includes the invocation of system workflows, such as restarting environments and triggering warm-up snapshots. To maintain test isolation and ensure repeatability, test cases can also include teardown definitions that describe post-condition expectations and cleanup actions. This guarantees that pods are terminated, runtime data is cleaned up, and residual state is removed between test runs.

The **validation suite** executes structured checks derived from the test definition, monitoring logs, resource status, and external APIs to confirm system correctness. Meanwhile, the test results and runtime logs are processed by the **result reporter** for test report generation. The **test controller** coordinates teardown actions based on execution outcomes and policy configurations.

Test Oracle. A test oracle is a principle that determines whether the observed output of a system under test for a given input is correct, fundamentally addressing the decidability problem in program verification (22). For this test suite, the oracle contains expected system properties, remote storage status, and runtime states of Kubernetes primitives, which will be verified after test execution.

4.5 Test Case Categories

This section presents the test suite in detail, organizing test cases around three core validation use cases derived from the overall validation goals. Component functionality correctness (Section 4.5.1) targets the validation of two core CodeCanvas workflows: the development environment lifecycle **(VG1)** and warm-up snapshots **(VG2)**. Data integrity (Section 4.5.2) focuses on verifying that persistent data is maintained correctly across version upgrades **(VG3)**. Cross-version compatibility of components (Section 4.5.3) evaluates whether system components remain interoperable under newer deployments **(VG4)**.

Given the complexity of CodeCanvas workflows and their orchestration across distributed Kubernetes components, we adopt a **dual-coverage strategy** to guide the design of the test suite. *Structural coverage* ensures that core user workflows and components are functionally validated, while *operator command coverage* provides evidence that all relevant backend logic is exercised in the Kubernetes Operator and is covered by the test suite, which is captured through a matrix of *operator commands* (create, delete, list, etc.) across *resource types* (job, pod, etc.). Unlike code coverage, which is difficult to measure in distributed systems (23), operator command coverage validates that the test suite not only aligns with high-level workflows but also exercises the essential logic embedded in the operator. Together, the two perspectives provide confidence that our validation addresses both expected and edge-case behavior, as well as the underlying mechanisms implementing them.

To maximize efficiency without compromising thoroughness, the test cases are designed according to a **minimal–maximal principle** (24): a minimal set of cases is selected such that it achieves maximal coverage under both structural and operator command metrics. This design choice reduces redundancy, shortens execution time, and limits maintenance overhead.

To approximate the coverage of the large input configuration space for each use case, **random input generation** is used as a representative sampling technique. The resulting operator coverage serves as an upper bound of achievable command–resource combinations under the given scenario. Therefore, the operator coverage of the final test cases should preserve all command–resource pairs observed in the sampled space while avoiding unnecessary duplication, thus reducing testing resources without sacrificing the coverage of validation.

4.5.1 Use Case 1: Component Functionality Correctness

This use case addresses the core functional behavior of the CodeCanvas application and its orchestration of remote development environments in a Kubernetes-based infrastructure. It verifies

Table 4.2: Configuration space of UC1 - component functionality correctness validation

Template Configuration	Compute instance type, IDE type, IDE version and IDE settings, SSH configuration, Environment variables, Standby pool, Development environment lifecycle scripts, Personal parameters, Cloud policy
Warm-up Configuration	Presence, Warm-up scripts, Warm-up parameters
User Roles	User access

that each infrastructure component involved in the development environment and warm-up lifecycle continues to function as expected after a Kubernetes cluster update. Since core infrastructure components are tightly coupled to specific workflow stages (Section 4.1), validating workflow execution correctness directly validates component correctness and their integration, addressing **VG1** and **VG2**.

To systematically validate this behavior, we translate component responsibilities into expected workflow transitions and verify them post-update. The test cases in this category focus on executing primary workflows of the development environment lifecycle and observing their successful transitions, such as creation, activation, and deletion. These observed transitions are then validated against a formal **finite state machine** shown in Figure 4.2 that encodes the correct sequence of environment states. This modeling ensures that the testing oracles are correctly calculated through different combinations of user actions. This model ensures that testing oracles are derived consistently across various combinations of user actions. Inputs include the creation and deletion of configuration templates, warm-up snapshots, and development environments, with failure states also considered.

Furthermore, at the component level, the test cases ensures that the **Kubernetes Operator** invokes the correct APIs, and that the backend orchestration across components like the **CodeCanvas Application**, **Jump Server**, and **Relay Server** remains operational. Log availability is also assessed to ensure traceability and remote storage accessibility.

Configuration Space. The configuration space represents the input domain from which test instances are derived. It defines the structural setup of each test case, influencing both the data created during test initialization and the functional pathways exercised during test execution.

For this use case, each execution path of a development environment workflow is parameterized by the configuration space that influences both component behavior and their interactions. Thus, to comprehensively validate the correctness of components and their orchestration, it is essential

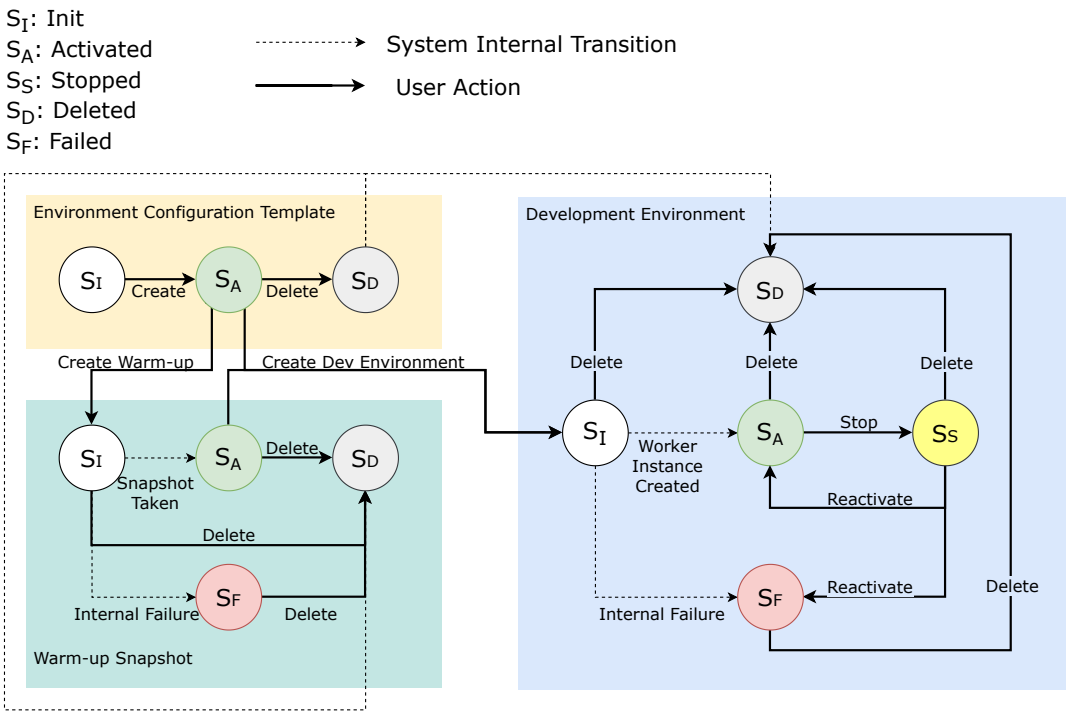


Figure 4.2: Finite state machine of development environment workflow

to assess system behavior across a representative span of the configuration space. As shown in Table 4.2, the configurations represent key dimensions that influence the behavior of components during the development environment lifecycle.

Test Cases. Table 4.7 outlines the test cases designed to validate component functionality correctness following a Kubernetes cluster upgrade. Each test targets a critical part of the development environment (*dev. env.*) lifecycle. **TC-01** exercises the warm-up and environment creation flow, verifying end-to-end orchestration and template reuse. **TC-02** checks the accessibility and persistence of logs for development environments that were created prior to the upgrade, while **TC-03** performs a similar validation for warm-up snapshots.

Structural Coverage. Implementing **VG1** (development environment lifecycle correctness) and **VG2** (warm-up snapshot validity), the initial test design is informed by core workflow models derived from relevant lifecycle transitions (Figure 4.2), such as creating, restarting, and deleting development environments and warm-up snapshots. These workflows are then encoded as testable

Table 4.3: Test cases of component functionality correctness validation

ID	Description	Preparation	Execution	Test Oracle
TC-01	New warm-up and dev. env. creation.	Create a template configuration and then create a warm-up snapshot of the template via API.	Recreate warm-up with the template and create a new dev. env. with that new warm-up.	The environment is active, the warm-up is overwritten.
TC-02	Log access of pre-created dev. env.	Create a template configuration, create and stop a dev. env. via API.	Open the pre-created dev. env. and download environment logs.	The logs are persistent and accessible.
TC-03	Log access of pre-created warm-up snapshot.	Create a template configuration, create a warm-up snapshot of the template via API.	Open the pre-created warm-up snapshot and download its logs.	The logs are persistent and accessible.

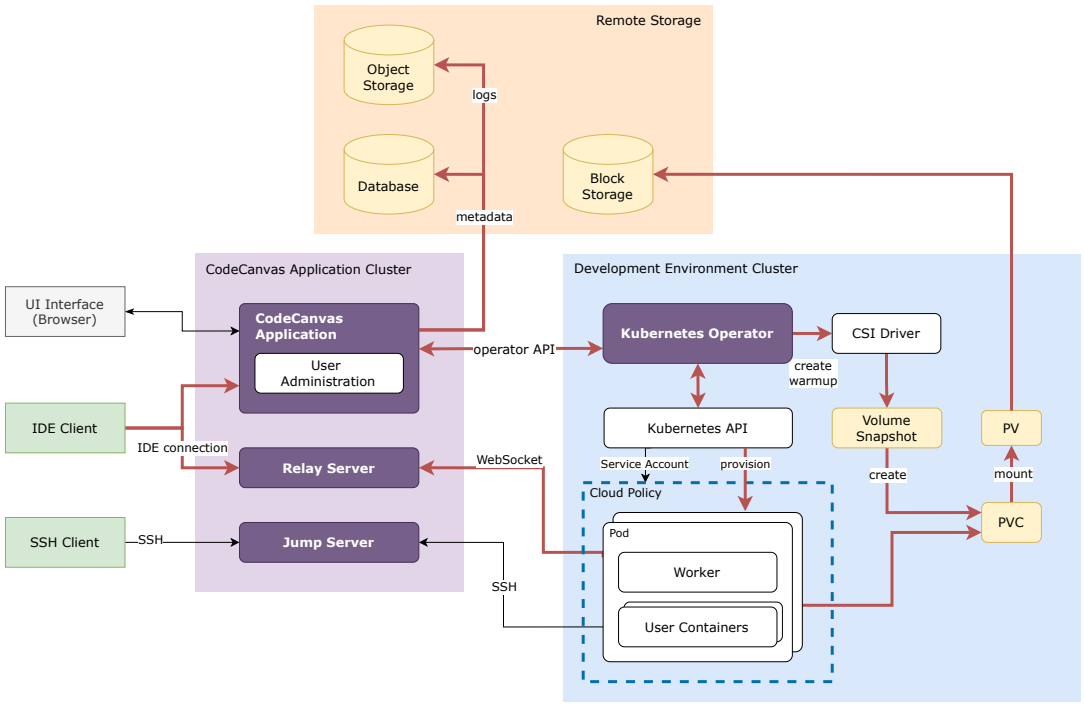


Figure 4.3: Structural coverage of test cases in UC1

sequences involving user actions on environment templates, warm-up states following the configurations in Table 4.2.

As shown in Figure 4.3, the test cases cover the red outlined interactions between core components. **TC-01** validates the full initialization flow, from template definition to warm-up snapshot recreation and development environment instantiation, covering transitions through key runtime states (*e.g.*, `Created`, `Activated`, `Stopped`). **TC-02** and **TC-03** cover the data access flow from the CodeCanvas Application Cluster, including user data in the database and the environment logs stored in object storage.

Operator Command Coverage. To bridge the gap between workflow behavior and backend implementation, the test suite design follows component analysis in Section 4.1, in which internal components are exercised by each workflow. Since the main workflow implementation is transferred by the Kubernetes operator, as shown in Figure 4.1, we adopt operator command coverage as a structured coverage model through a matrix of *operator commands* across *resource types*, where each **row** corresponds to a specific operator command and each **column** corresponds to a resource type managed by the Kubernetes Operator. Each cell in the matrix contains the number of times that the corresponding command–resource pair was invoked during test execution. The values are represented both numerically and through a heat map color scale, where higher frequencies are shown in lighter shades.

Figure 4.4 shows the coverage upper bound of the operator command matrix through random input generation. Respectively, Figure 4.5 shows the operator command coverage from the input of the test cases for use case 1. Both matrices have identical command–resource combinations, indicating that the designed test cases achieve complete coverage of the command–resource pairs exercised by the random sampling approach. This result validates the application of the *minimal–maximal* principle, demonstrating that the reduced set of deterministic test case inputs preserves the implementation-level coverage.

4.5.2 Use Case 2: Data Integrity

This use case targets the reliability of persistent data across Kubernetes cluster updates. It focuses on verifying that both user-generated artifacts (*e.g.*, template configurations) and system-generated metadata (*e.g.*, logs, warm-up snapshots) remain intact, accessible, and correctly associated with their owning entities after the upgrade. These checks directly address **VG3**, which emphasizes guarantees on persistent data integrity during version transitions.

Data storage in CodeCanvas may be recreated, remounted, or re-accessed after a Kubernetes cluster update, such as PVCs, object storage, and block storage. Test cases for this use case systematically exercise workflows that access or rely on pre-existing data artifacts created prior to the cluster update event. These include restarting environments, accessing warm-ups and logs,

Figure 4.4: Operator command coverage of random input generation from UC1 configuration space

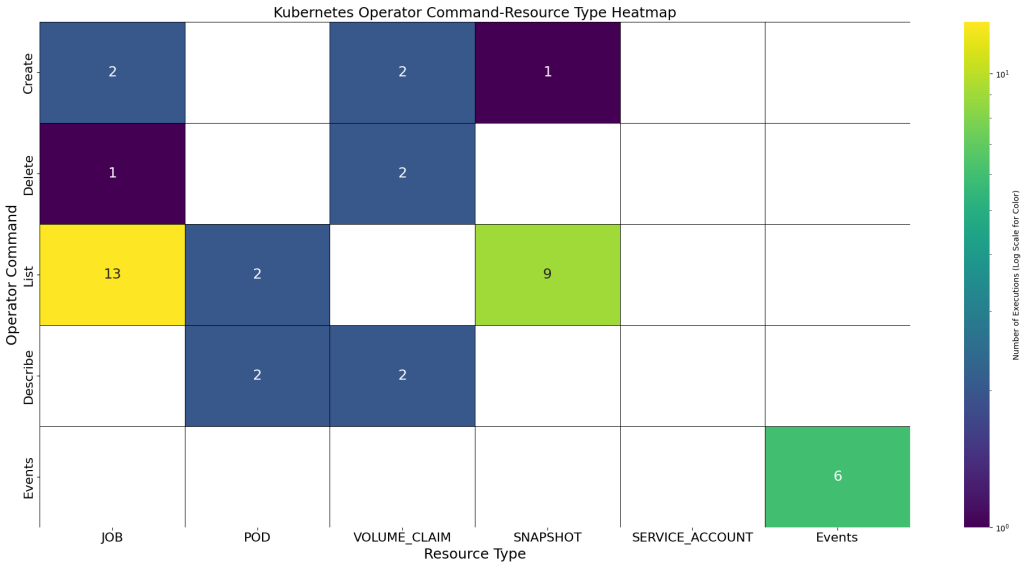


Figure 4.5: Operator command coverage from UC1 test cases

and interacting with policies or roles that encode permissions. Each test case confirms that no regression, data loss, or misconfiguration failure occurs as the system resumes operation on the upgraded cluster.

Configuration Space. The test cases are derived from a structured configuration space that defines the input domain for test execution, which captures all relevant parameters that influence

Table 4.4: Configuration space of data integrity

Template Configuration	Compute instance type, IDE type, IDE version and IDE settings, SSH configuration, Environment variables, Standby pool, Development environment lifecycle scripts, Personal parameters, Cloud policy
Warm-up Configuration	Presence, Warm-up scripts, Warm-up parameters
User Roles	User access to template configurations and dev. env.
Cloud Policy	Corresponding Kubernetes operator, Relay server, Jump server, Kubernetes service account
Feature Flag	User visibility (everyone, certain user groups, certain users)

what data is created, how it is stored, and how it is expected to behave after the upgrade.

As shown in Table 4.4, each configuration option contributes to potential variance in test outcomes. For instance, warm-up presence and parameters affect snapshot generation and reuse logic. User roles and cloud policies encode access constraints that impact data visibility and modification rights.

Test Cases. As shown in Table 4.5, by covering a range of operations that interact with persistent volumes, warm-up snapshots, and logs, and by executing related workflows, the test suite validates the system's ability to preserve data consistency and correctness across upgrade boundaries.

Coverage. From a data integrity point of view, the test cases cover **all** storage units in remote storage. **TC-02** and **TC-03** cover logs access in *object storage*. **TC-06** covers the access of *block storage*. The rest of the test cases cover different data types of system metadata stored in the *database*, including user data and development environment configurations.

Unlike the functionality correctness use case, where verifying the full range of operator resource-command combinations is essential, operator command coverage is not a primary focus in the data integrity context. While operator routines are indirectly exercised by test cases, persistent data is validated through end-to-end outcomes rather than by explicitly tracking operator actions.

4.5.3 Use Case 3: Cross-Version Component Compatibility

This use case focuses on validating the compatibility between different versions of Kubernetes-deployed infrastructure components and the CodeCanvas application under continuous integration and delivery. For instance, after a cluster upgrade, only the CodeCanvas application component

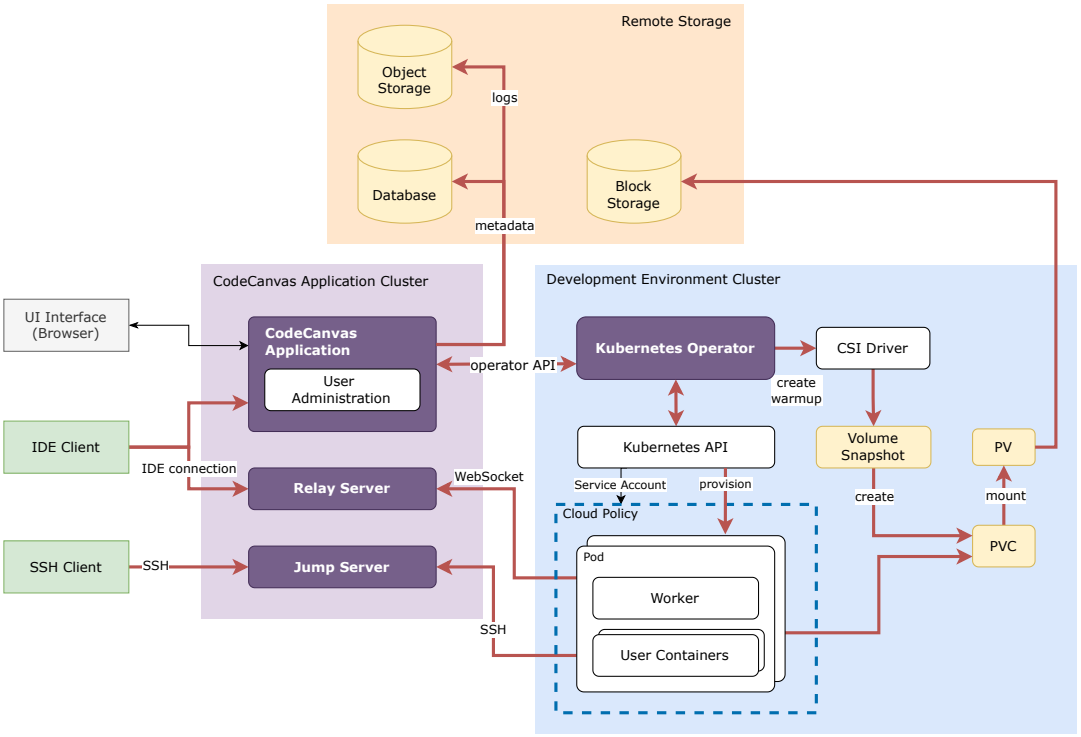


Figure 4.6: Structural coverage of test cases in UC3

is re-deployed due to changes, while previously deployed versions of the operator, relay server, and jump server remain active, which emphasizes the need for compatibility testing to ensure coordination and orchestration correctness across component versions, directly addressing **VG4** (cross-version compatibility of core components).

Configuration Space. Similar to use case 1, Table 4.6 lists the configuration space for this use case scenario, which covers not only all the inputs that request application and native Kubernetes components but also the inputs that invoke cross-component interactions.

Test Cases. The test cases capture the possible interactions between CodeCanvas internal components. **TC-01** covers the interaction between Kubernetes operator and CSI Driver through warm-up snapshot creation post-update. **TC-04 to 06** cover the interactions related to the Relay Server. **TC-10** covers the validation of Jump Server interactions with other components through SSH operations.

Figure 4.7: Operator command coverage of random input generation from UC3 configuration space

Structural Coverage. To validate compatibility across versions of CodeCanvas components (CodeCanvas application, Operator, Relay, and Jump server), we focus on integration-critical workflows that require inter-component communication. As shown by the red outlined arrows in Figure 4.6, all the interactions of CodeCanvas components are covered by the test cases for this use case scenario.

Operator Command Coverage. Rather than testing every possible configuration mutation, we use the operator command coverage matrix to guarantee that all relevant behaviors from the configuration space are exercised. As shown in Figure 4.7 and Figure 4.8, the two matrices contain the same set of command–resource pairs, confirming that the designed test cases fully cover the operator-level input space from random sampling.

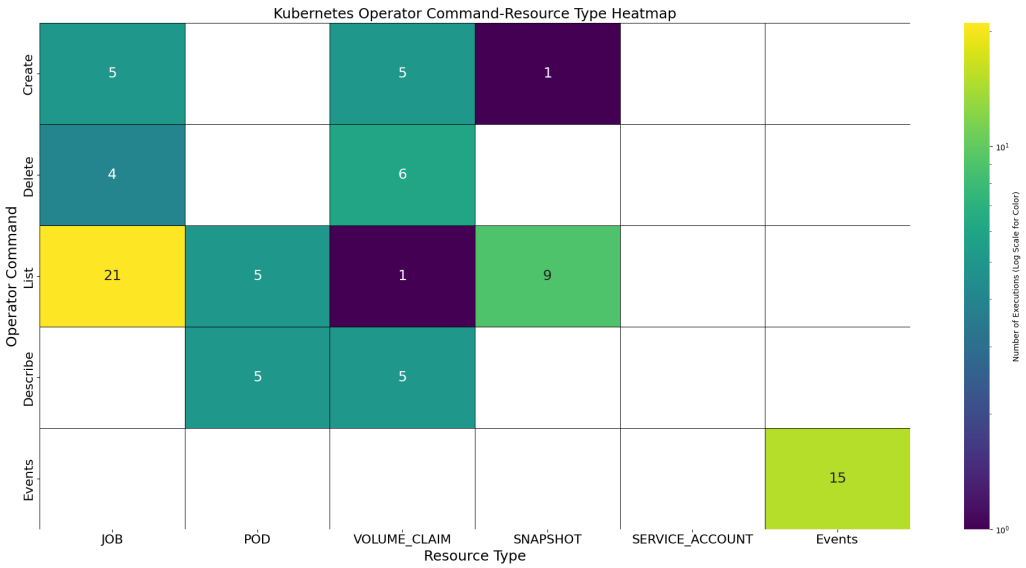


Figure 4.8: Operator command coverage from UC3 test cases

Table 4.5: Test cases of data integrity

ID	Description	Preparation	Execution	Test Oracle
TC-02	Log access of pre-created dev. env.	Create a template configuration, create and stop a development environment via API.	Open the pre-created dev. env. and download environment logs.	The logs are persistent and accessible.
TC-03	Log access of pre-created warm-up snapshot.	Create a template configuration, create a warm-up snapshot of the template via API.	Open the pre-created warm-up snapshot and download its logs.	The logs are persistent and accessible.
TC-04	Reactivate pre-created dev. env.	Create a template configuration, create and stop a dev. env. via API.	Reactivate the pre-created environment.	The environment is active.
TC-05	Reactivate pre-created dev. env. with warm-up.	Create a template configuration, create a warm-up snapshot of the configuration via API, create and stop a dev. env. using the warm-up.	Reactivate the pre-created environment.	The environment is active.
TC-06	Create new dev env using pre-created warmup.	Create a template configuration, create a warm-up snapshot of the configuration via API.	Create a new dev. env. via API with the pre-created warm-up configuration.	The environment is created and active. The warm-up snapshot is valid and not overwritten.
TC-07	Cloud policy created before cluster update is usable post-update.	Create a new cloud policy via UI.	Create a template configuration using the pre-created cloud policy, create a new dev. env. with the template.	The cloud policy persists. The environment is created and active under the policy.
TC-08	User access configuration is persistent after update.	Configure a custom user role with certain set of permissions and create a new user under that role.	Log in as the new user and access CodeCanvas.	The user permission matches the role configuration.
TC-09	Feature flag configuration is persistent after update.	Modify feature flag configurations as: enable one feature flag for everyone and enable another feature flag for a custom user.	Access feature flag configuration.	The feature flag configuration modifications persist.

Table 4.6: Configuration space of cross-version component compatibility

Template Configuration	Compute instance type, IDE type, IDE version and IDE settings, SSH configuration, Environment variables, Standby pool, Development environment lifecycle scripts, Personal parameters, Cloud policy
Warm-up Configuration	Presence, Warm-up scripts, Warm-up parameters
User Roles	User access

Table 4.7: Test cases of cross-version component compatibility validation

ID	Description	Preparation	Execution	Test Oracle
TC-01	New warm-up and dev. env. creation.	Create a template configuration and then create a warm-up snapshot of the template via API.	Recreate warm-up with the template and create a new dev. env. with that new warm-up.	The environment is active, the warm-up is overwritten.
TC-04	Reactivate pre-created dev. env.	Create a template configuration, create and stop a dev. env. via API.	Reactivate the pre-created environment.	The environment is active.
TC-05	Reactivate pre-created dev. env. with warm-up.	Create a template configuration, create a warm-up snapshot of the configuration via API, create and stop a dev. env. using the warm-up.	Reactivate the pre-created environment.	The environment is active.
TC-06	Create new dev. env. using pre-created warmup.	Create a template configuration, create a warm-up snapshot of the configuration via API.	Create a new dev. env. via API with the pre-created warm-up configuration.	The environment is created and active. The warm-up snapshot is valid and not overwritten.
TC-10	Reactivate pre-created dev. env. and test SSH access.	Create a template configuration with SSH configured, create and stop a dev. env. via API.	Reactivate the pre-created environment. Attempt to connect the environment using SSH commands.	The environment is created and active. SSH connection is successful.

5

Implementation

This section illustrates the practical implementation of the testing framework and the test suite.

5.1 Implementation of the Testing Framework

Following the design principles outlined in Section 3, the testing framework is implemented as an integrated pipeline within *JetBrains TeamCity* CI/CD platform, meeting the operational requirements of CodeCanvas, which is a large-scale Kubernetes-based remote development system undergoing frequent updates. The framework consists of five core components: the test controller, cluster manager, test data injector, validation suite, and result reporter, which simulate real-world version update scenarios.

Figure 5.1 depicts the detailed architecture of these components and their interaction at various stages of the testing workflow. The **test controller** is implemented using the Kotlin DSL[1], which allows build configurations to be stored in version control and supports declarative orchestration of build pipelines. The test controller specifies the execution order, environment setup, and dependency handling between stages, ensuring reproducibility across test executions.

The **cluster manager** consists of scripts for both Kubernetes cluster initialization and cluster upgrades, using `helm`[2] to manage CodeCanvas deployments. It provisions clusters from Helm charts and applies Helm chart changes corresponding to target versions. The **test data injector** and **validation suite** are both implemented using JUnit 5[3], but differentiated by tagging strategies. Data preparation tests are tagged with `"Data"`, while post-upgrade validation tests use `"Suite"`, allowing the test controller to run only the relevant subset of tests at each stage of the workflow. The test artifacts from the data injection and validation suite are gathered, processed, and presented

[1] Kotlin DSL in TeamCity: `https://www.jetbrains.com/help/teamcity/kotlin-dsl.html?Kotlin+DSL`
[2] Helm, a pacakge manager for Kubernetes: `https://helm.sh/`
[3] JUnit 5 testing framework for JVM: `https://junit.org/`

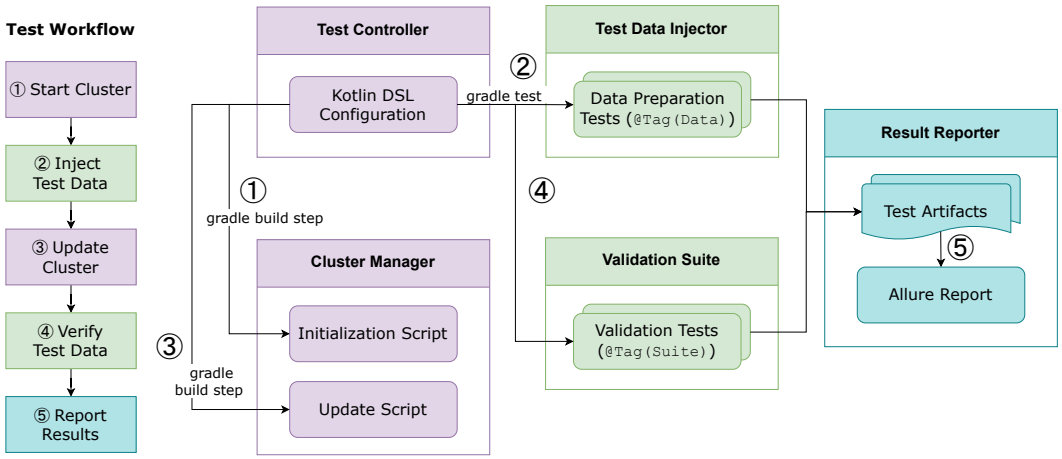


Figure 5.1: Implementation of the testing framework

by the result reporter, which is implemented using Allure Report[1], with an interactive dashboard summarizing pass and fail statistics.

Execution begins when the test controller triggers the cluster manager's initialization step for starting a cluster (1) with a default version, which is executed as a Gradle build task. After the cluster is provisioned, the controller executes the data preparation tests (2) by running Gradle test command on tests tagged for data injection. The controller then initiates the upgrade phase by invoking the cluster manager's update scripts (3), applying a Helm release to transition the cluster from the source to the version-to-be-tested. Once the update is complete, the validation suite is executed (4), with the controller filtering for validation test cases. Finally, all the test artifacts are processed and published at the end of the execution (5).

5.2 Implementation of the Test Suite

5.2.1 Operator Command Coverage Calculation

To estimate the operator command coverage attauc1-ramdinable within each use case, we implement a random input generation process that samples from the input space of valid user actions as described in Section 4.5. For each use case scenario, the input configuration is organized into modular action sequences, which are randomly invoked 20 times on a local CodeCanvas infrastructure instance, balancing local instance capacity and performance.

During each execution, the test framework performs the corresponding actions in a controlled test cluster and records all operator commands invoked by the Kubernetes Operator, along with

[1] Allure report, HTML test automation report tool: `https://allurereport.org/`

their associated resource types, as captured in the operator logs. The collected logs are processed using a Python analysis script, which extracts command–resource pairs and computes the resulting operator command coverage. This data was then visualized as a command–resource matrix heat map, generated with the `pandas` and `matplotlib.pyplot` libraries.

5.2.2 Test Case Implementation

The test suite is implemented as a modular collection of automated tests, following the design outlined in Section 4. It is developed in Kotlin using JUnit 5, with Gradle as the build and execution engine, ensuring seamless integration with the testing framework described in Section 3.

Each test case follows the structure defined in Table 4.1, consisting of three main components: test preparation, execution, and test oracle. Test preparation injects the necessary initial test data into the system, execution performs the operation steps under evaluation, and the test oracle verifies the resulting system state against expected outcomes. Metadata for each test case is maintained in the test controller, enabling controlled orchestration and selective execution.

Implementation-wise, test preparation methods are annotated with `@Tag("Data")`, while execution and oracle methods are annotated with `@Tag("Suite")` as described in the implementation of the testing framework. The test methods share a set of utility classes implemented as lightweight internal libraries, which encapsulate recurring operations such as invoking CodeCanvas APIs, interacting with the web UI, and parsing application logs.

The input to a test case is the data provisioned during the preparation phase, typically consisting of user metadata, Kubernetes persistent volumes, or volume snapshots. The output is the validation result produced by the oracle, which includes structured reports and cluster logs. These artifacts are collected by the result reporter for aggregation and publication.

Adding a new test requires creating a new class in the relevant package, implementing the preparation, execution, and oracle components by invoking existing testing modules or implementing relevant APIs, and applying the appropriate tags. Because common orchestration logic is encapsulated in utilities, new tests can be added with minimal effort, focusing primarily on the specific validation logic.

6

Evaluation

To answer *RQ3: How can the effectiveness of the designed testing framework and test suite be evaluated?*, this section presents the experimental validation of the testing framework designed in Section 3, structured around the three experiments corresponding to three primary use cases introduced in Section 4: component functionality correctness (UC1), data integrity (UC2), and cross-version component compatibility (UC3).

The experiments are executed in real deployment environments using production-grade test workflows. The testing framework simulates a CodeCanvas deployment update for end-users via the framework components, *cluster manager* , and *test controller*. For each test, we verify system outcomes and compare them against the expected states and properties from the model to assert correctness post-upgrade. Each experiment addresses a dedicated **evaluation research question** and maps onto a subset of test cases, as mentioned in Section 4.

- **Q1: Does CodeCanvas function correctly after a cluster upgrade across all critical infrastructure components?**
  This question aligns with UC1, focusing on validating the correctness of the components after updates are made to the system cluster by executing the main system workflows that cover the invocation of core components. (Section 6.2)
- **Q2: Does data stored in remote storage remain valid and accessible after a cluster upgrade?**
  This experiment creates different types of data artifacts prior to the upgrade, such as warm-up snapshots, logs, and certain configurations, and then verifies their availability and correctness post-upgrade, aligning with UC2. (Section 6.3)
- **Q3: Can the CodeCanvas core components be safely upgraded across versions while preserving both deployment correctness and runtime compatibility?**

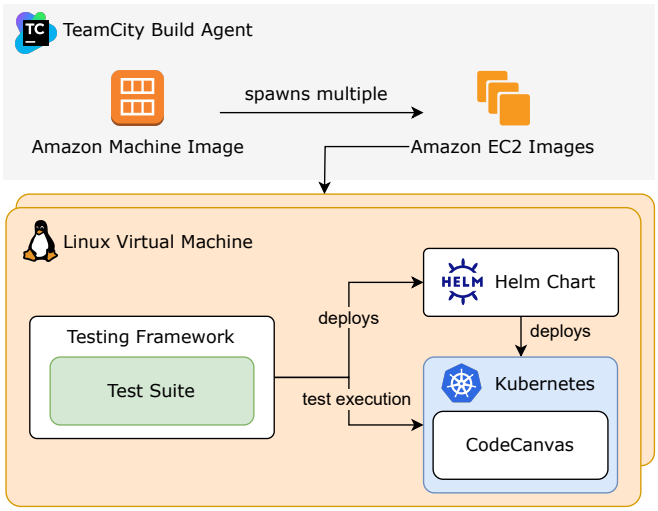


Figure 6.1: Experiment deployment pipeline

We evaluate cross version compatibility by mixing different versions of the Operator, Jump Server, Relay, and Application components in realistic development scenarios. In parallel, we validate that Helm chart upgrades correctly modify Kubernetes resources without introducing configuration errors. (Section 6.4)

The testing framework and test suite are fully integrated into the CodeCanvas development pipeline. For every proposed change to the CodeCanvas code base, the entire test suite is automatically executed prior to merge approval with the modified CodeCanvas cluster as the target version-to-be-tested, ensuring that no change is integrated unless all tests pass successfully. In the production environment, this process is triggered an average of 20 times per day.

For the evaluation of the three use case scenarios, three targeted experiments are designed to address the corresponding experimental research questions. Five distinct CodeCanvas versions, each containing real-world production changes to different system components, are selected as evaluation targets. The complete test suite is executed on each version, and the outcomes of all tests are recorded. To reduce the impact of environmental variability and test flakiness, each test case for a given target version is executed five times, with results aggregated for analysis. A test case is classified as `failed` if it fails in all five executions.

Table 6.1: Experiment environment configuration

Operating System	Ubuntu 22.04.5 LTS
Kernel Version	6.8.0-1029-aws
CPU Model	AMD EPYC 7R13, 8 cores × 2 threads (16 threads total)
Architecture	x86_64
Memory (RAM)	32GB

6.1 Experiment Setup

The experimental setup is designed for scalability using a cloud-based CI/CD tool, *JetBrains TeamCity*. As shown in Figure 6.1, experiments are executed on TeamCity build agents running on Amazon EC2 instances, which are provisioned from a custom **Amazon Machine Image** (AMI) configuration to ensure a consistent environment across runs. Each EC2 instance hosts a **Linux-based virtual machine**, within which the entire test framework and test suite are deployed. The test framework is responsible for orchestrating the deployment of Codecanvas **Helm charts**, which in turn triggers the creation of **Kubernetes clusters** using Helm as the package manager. Once the clusters are provisioned, the test suite is executed directly inside the Codecanvas environment running within the Kubernetes clusters. This pipeline allows for end-to-end testing in isolated, reproducible environments that simulate real-world deployment scenarios. The Linux virtual machine configuration is listed in Table 6.1.

6.2 Experiment 1: Component Correctness

This experiment evaluates *Q1: Does CodeCanvas function correctly after a cluster upgrade across all critical infrastructure components?* The goal is to verify that the system preserves its expected operational behavior throughout the full development environment lifecycle as the underlying infrastructure components undergo changes.

To answer this question, we automatically executed the test cases mentioned in Section 4.5.1 that represent key workflows. From an end-to-end point of view, each test execution not only has a set of expected system behavior outcomes but is also grounded in a FSM model of expected environment transitions and states. The test results are validated post-upgrade to confirm alignment with expected states and transitions.

Each test is run in an isolated namespace to simulate real user environments. First, the environment is prepared in a pre-upgrade state by programmatically creating the necessary resources

Table 6.2: Test results of experiment 1

ID	Description	Test Oracle	Test Result
TC-01	New warm-up and dev. env. creation.	The environment is active, the warm-up is overwritten.	Pass (5/5)
TC-02	Log access of pre-created dev. env.	The logs are persistent and accessible.	Pass (5/5)
TC-03	Log access of pre-created warm-up snapshot.	The logs are persistent and accessible.	Pass (5/5)

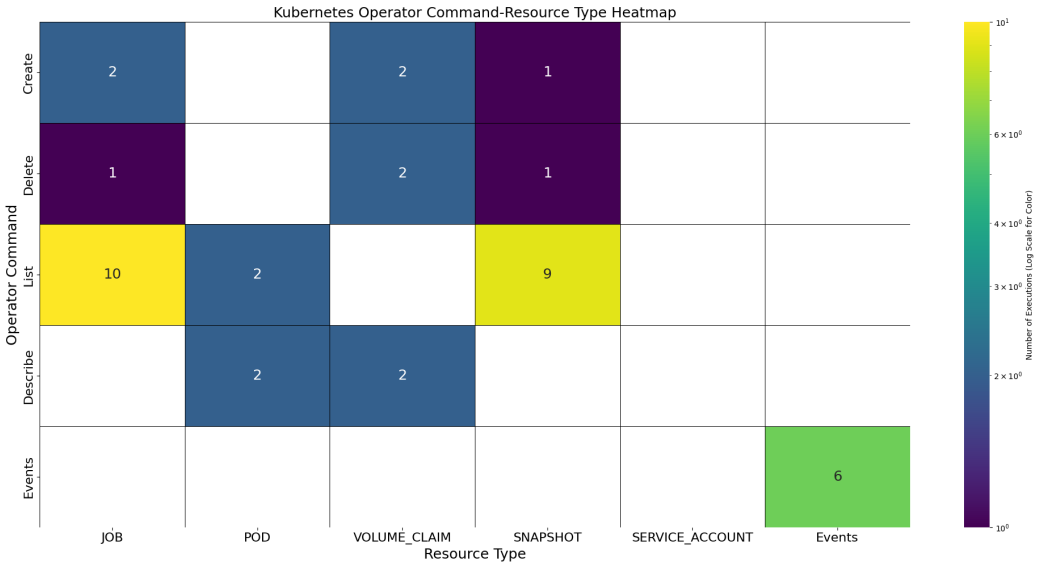


Figure 6.2: Operator command coverage of test cases in experiment 1

via corresponding API and UI actions defined in Table 4.7. Next, a simulated cluster upgrade is performed by bumping the Helm chart version to trigger a new deployment. Once the cluster upgrade is complete, the corresponding post-upgrade actions are triggered (test execution), such as restarting a development environment or recreating a warm-up. The expected outcome is determined using a FSM oracle, which encodes valid state transitions and behaviors. Finally, the system's logs and outputs are reported and validated to ensure consistency with the expected behavior, confirming that all components function correctly and no errors were introduced during the upgrade process.

To justify the complete coverage of the test cases, we calculate the operator command coverage of this experiment in Figure 6.2, confirming that all critical resource-command combinations for Use Case 1 are exercised during the tests.

Table 6.3: Test results of experiment 2

ID	Description	Test Oracle	Test Result
TC-02	Log access of pre-created dev. env.	The logs are persistent and accessible.	Pass (5/5)
TC-03	Log access of pre-created warm-up snapshot.	The logs are persistent and accessible.	Pass (5/5)
TC-04	Reactive pre-created dev. env.	The environment is active.	Pass (5/5)
TC-05	Reactivate pre-created dev. env. with warm-up.	The environment is active.	Pass (5/5)
TC-06	Create new dev env using pre-created warmup.	The environment is created and active. The warm-up snapshot is valid and not overwritten.	Pass (5/5)
TC-07	Cloud policy created before cluster update is usable post-update.	The cloud policy persists. The environment is created and active under the policy.	Pass (5/5)
TC-08	User access configuration is persistent after update.	The user permission matches the role configuration.	Pass (5/5)
TC-09	Feature flag configuration is persistent after update.	The feature flag configuration modifications persist.	Pass (5/5)

Overall, this experiment confirms that all primary workflows operate correctly post-upgrade, with no regressions or broken state transitions. These results support the conclusion that CodeCanvas's core components remain functionally correct across Kubernetes upgrades, positively answering Q1.

6.3 Experiment 2: Data Integrity

This experiment targets Q2: Does data stored in remote storage remain valid and accessible after a cluster upgrade? The focus is on verifying the durability and consistency of persistent storage elements such as development environment volumes, warm-up snapshots, and logs across upgrade boundaries.

To test this, we create a set of controlled data artifacts prior to a simulated cluster upgrade, including active environments, stopped environments, and warm-ups. After the upgrade, we reaccess these artifacts and performed operations such as restarting environments, downloading logs, and recreating warm-ups from saved templates. Similar to experiment 1, each test is validated

using outcome oracles that check for the presence of expected files, the consistency of the environment state, and the availability of log content. Unlike Use Case 1, operator command coverage is not a relevant metric here, as these tests did not aim to exhaustively test orchestration behaviors but rather to ensure data continuity.

The test results are recorded in Table 6.3. All test cases passed, and data remained accessible and unmodified after upgrade operations. This includes object storage logs, PVC-mounted workspace volumes, and metadata artifacts such as warm-up parameters. These results strongly suggest that CodeCanvas preserves user and system data correctly through cluster upgrades, positively answering Q2.

6.4 Experiment 3: Version Compatibility

This experiment evaluates Q3: Can the CodeCanvas core components be safely upgraded across versions while preserving both deployment correctness and runtime compatibility? The goal is to determine whether system functionality remains intact when certain components are updated independently, simulating real-world deployment patterns in which infrastructure elements are updated separately in different versions.

To systematically answer this question, we construct a two-phase deployment setup with the proposed testing framework. First, we initialized a CodeCanvas cluster with older Helm chart versions and provision user environments and warm-ups to generate persistent system state. Then, we upgrade one or more components using Helm, transitioning to newer chart versions. Each upgrade scenario is followed by a sequence of workflow re-executions (*e.g.*, environment restart, warm-up recreation) to validate continued compatibility between upgraded and legacy components.

Test coverage is assessed using operator command coverage, which measures the extent to which the test cases exercise the set of Kubernetes resources managed by the Kubernetes Operator within the configuration space. As shown in Figure 6.3, all command–resource pairs defined in the configuration space of Use Case 3 are covered by the test executions.

As shown in Table 6.4, test results indicate that all major workflows including environment reactivation and warm-up access function correctly across component version boundaries suggesting that upgrade safety holds for critical workflows, affirmatively answering Q3.

6.5 Bug Example

During daily production, the testing framework and test suite have successfully identified bugs in CodeCanvas development. In one instance, a Helm chart included a conditional statement: `{{-`

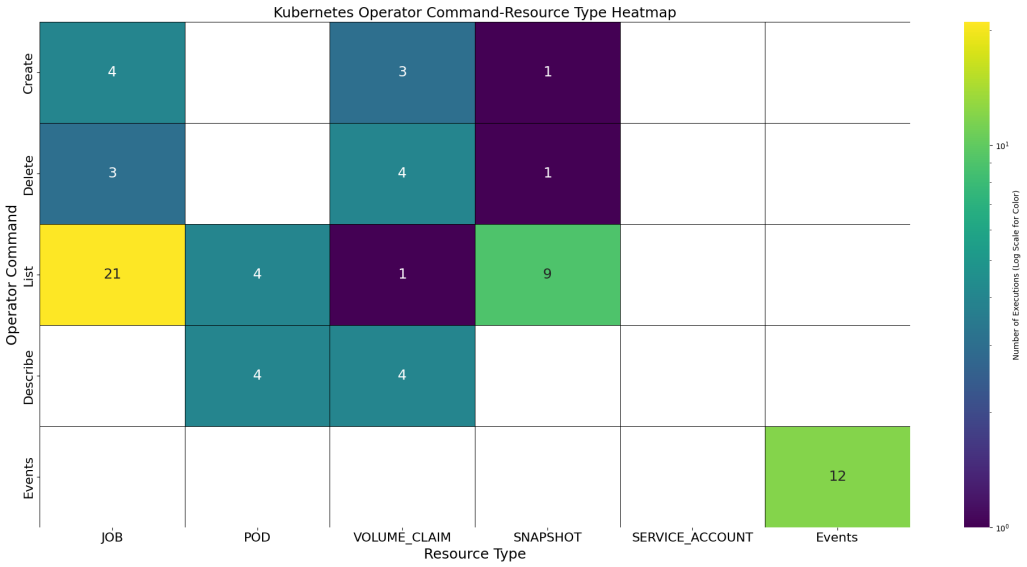


Figure 6.3: Operator command coverage of test cases in experiment 3

`if .Release.IsInstall}}`, which creates a Kubernetes secret during the initial installation. However, when the same chart was applied via a Helm upgrade during a cluster update in our framework, the condition evaluated to `false`, causing the secret to be skipped and the update to fail. This behavior illustrates a Helm-specific upgrade nuance that is not captured by standard test executions in isolated environments using a fresh installation.

Table 6.4: Test results of experiment 3

ID	Description	Test Oracle	Test Result
TC-01	New warm-up and dev. env. creation.	The environment is active, the warm-up is overwritten.	Pass (5/5)
TC-04	Reactivate pre-created dev. env.	The environment is active.	Pass (5/5)
TC-05	Reactivate pre-created dev. env. with warm-up.	The environment is active.	Pass (5/5)
TC-06	Create new dev. env. using pre-created warmup.	The environment is created and active. The warm-up snapshot is valid and not overwritten.	Pass (5/5)
TC-10	Reactivate pre-created dev. env. and test SSH access.	The environment is created and active. SSH connection is successful.	Pass (5/5)

7

Related Work

Testing large-scale Kubernetes-integrated systems like CodeCanvas presents unique challenges, particularly when ensuring both functionality correctness of the application and native Kubernetes components as well as compatibility across system updates during continuous delivery cycles. This section reviews existing research in three key areas relevant to our work: (1) testing container orchestration systems, (2) upgrade and cross-version compatibility testing, and (3) end-to-end testing methodologies. Each subsection highlights current approaches, their limitations, and how our testing solution for CodeCanvas addresses the identified gaps.

7.1 Testing Container Orchestration Systems

Research on testing container orchestration systems, such as Kubernetes and Docker Swarm, has largely focused on functional validation of their native orchestration features (9, 25, 26)and applications that are deployed within such systems (11). In addition, prior work has investigated integration testing techniques targeting both distributed applications and the underlying Kubernetes control plane (27, 28). However, these approaches frequently rely on single-cluster setups and synthetic workloads, which may not fully capture the complexity and variability of production-scale deployments.

In contrast, the framework proposed in this thesis conducts functional validation across multiple layers, capturing both the native Kubernetes components and the system's customized orchestration logic. Furthermore, it evaluates system behavior under realistic, production-like workloads in real-world contexts.

7.2 Cross-Version Compatibility Testing

Maintaining compatibility across versions of software components is a persistent challenge in modern systems. Jayasuriya et al.(29) proposed a comprehensive taxonomy of version incompatibility changes that lead to update failures in Java projects, providing a systematic classification of breaking changes in system updates. McCamant et al. (30) identified incompatibilities among software components during real-world upgrades and implemented a framework to predict possible upgrade failures due to component versions. Focusing specifically on backward incompatibilities, Ruiz et al.(31) conducted large-scale regression testing on consecutive version pairs of widely used Java libraries, revealing 280 groups of behavioral incompatibilities that could not be detected through signature analysis alone. Complementing these studies, Horton and Parnin(32) introduced *V2*, a feedback-directed testing strategy designed to identify outdated code snippets or configurations during updates.

Existing research targets static libraries or single-component upgrades. However, our testing approach evaluates *multi-component compatibility* by simulating real-world end user updates under realistic cluster conditions and integrating automated upgrade testing into a continuous integration workflow.

7.3 End-to-end Testing

End-to-end testing is a software testing methodology that evaluates the complete functionality of a system in a production-like environment. Unlike unit testing, which targets individual functions or classes, and integration testing, which focuses on interactions between selected components, end-to-end testing assesses whether the system behaves as expected when all components interact under realistic operational conditions. In the context of Kubernetes-integrated systems, this includes the full lifecycle from system deployment and validation of system behavior to system termination.

Several end-to-end testing approaches have been developed for containerized and Kubernetes-based systems. Reile et al.(28) introduced *Bunk8s*, a microservices testing tool that integrates with CI/CD pipelines and container orchestration systems to launch and coordinate test runner containers in dedicated pods. Gu et al.(25) proposed *Acto*, an end-to-end testing framework that models Kubernetes Operators as state machines and systematically explores their state transitions to verify both operator correctness and the ability of managed systems to reach intended states. Maliekal (33) proposed an automated testing framework for system deployed in Kubernetes, in-

cluding a CI/CD platform integration which executes unit tests and static code analysis before cluster deployment.

Compared to these approaches, the testing framework and suite developed in this thesis extend the scope of end-to-end testing by validating not only native Kubernetes functionality but also customized orchestration logic and integrated services under real-world workloads. Furthermore, it integrates cross-version compatibility tests into the CI/CD pipeline, enabling early detection of upgrade-introduced regressions.

8

Conclusion

8.1 Answering Research Questions

Through a use-case study on the remote development platform *JetBrains CodeCanvas*, this thesis addresses *the lack of a systematic methodology and automated framework for end-to-end testing of Kubernetes-integrated systems* that both run as applications on Kubernetes and incorporate orchestration logic within the Kubernetes control plane.

We contribute a **methodology** for the design of an orchestration-aware, continuous end-to-end testing framework and suite, validated through the **design, implementation, and evaluation** of a concrete testing solution for CodeCanvas. While developed for CodeCanvas, the methodology and framework are generalizable to other Kubernetes-integrated systems, such as OpenShift-based platforms.

RQ1: How can we design a continuous orchestration-aware testing framework for Kubernetes-integrated systems? We answer this question by introducing a design methodology that begins with a system workflow analysis to identify relevant components, orchestration logic, and their interdependencies. This is followed by a functional and non-functional requirements analysis, ensuring coverage of both application-level and Kubernetes control plane behaviors. Based on this, we designed and implemented a testing framework for CodeCanvas that (i) incorporates orchestration-aware mechanisms to capture interactions between custom Kubernetes logic and core cluster components, (ii) supports validation across cluster upgrades, and (iii) integrates with the CI/CD pipeline for continuous execution.

RQ2: How can an effective test suite be constructed for Kubernetes-integrated systems? We propose a three-step methodology for the design of the test suite, including (i) a component

analysis of the system under test to identify the functional scope and integration points between application components and Kubernetes resources, (ii) the identification of design principles that define the structure of the test suite, and (iii) the derivation of validation goals that capture both functional correctness and orchestration-level guarantees. These validation goals are then systematically mapped to three representative use case scenarios, which are in turn transformed into concrete test cases. Each test case is designed to minimize input space while maximizing coverage, enabling efficient and thorough validation.

- **RQ2.1: Which components of a Kubernetes-integrated system must be tested?** The component analysis identifies both application-specific services and Kubernetes components that directly influence orchestration behavior. These components form the primary test scope.
- **RQ2.2: What are the expected behaviors and outcomes for different Kubernetes-based system components under various conditions?** We perform a component analysis on the CodeCanvas application and customize Kubernetes components, identifying the detailed behavior of each component and possible failures of the components. This forms the foundation for defining validation goals and constructing relevant use cases.

RQ3: How can the effectiveness of the designed testing framework and test suite be evaluated? We evaluate the framework and suite by integrating them with JetBrains TeamCity, the CI/CD pipeline hosting CodeCanvas. The framework was executed in repeated test runs across multiple CodeCanvas versions, each containing real-world modifications to application and Kubernetes-related components. Additionally, the test framework and suite are incorporated into the safe-merge pipeline, ensuring that every merge request triggers orchestration-aware validation. This integration led to the detection of bugs that were undetectable by conventional functional testing.

8.2 Limitations and Future Work

The proposed framework and test suite have proven to be effective for CodeCanvas, but their scope is still limited due to design trade-offs made to balance coverage and feasibility. The applicability of the testing framework to systems with different architectures or workloads has not yet been tested. Meanwhile, the test suite is designed with deterministic scenarios and minimal input configurations to achieve high coverage efficiently, but this may overlook rare or complex orchestration failures. Future work could apply the framework to a wider range of systems, including

fuzzing or chaos testing to detect non-deterministic issues, and introduce adaptive mechanisms to update test cases as orchestration logic or Kubernetes versions change.

References

[1] WES FELTER, ALEXANDRE FERREIRA, RAM RAJAMONY, AND JUAN RUBIO. **An updated performance comparison of virtual machines and Linux containers**. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pages 171–172. IEEE Computer Society, 2015. 1

[2] BRENDAN BURNS, BRIAN GRANT, DAVID OPPENHEIMER, ERIC A. BREWER, AND JOHN WILKES. **Borg, Omega, and Kubernetes**. *Commun. ACM*, **59**(5):50–57, 2016. 1

[3] CLOUDNATIVECOMPUTINGFOUNDATION. **Cloud Native Computing Foundation Annual Survey 2024** [cited 08.04.2025]. 1

[4] SALLA TIMONEN, MAHA SROOR, RAHUL MOHANANI, AND TOMMI MIKKONEN. **Anomaly Detection Through Container Testing: A Survey of Company Practices**. In REGINE KADGIEN, ANDREAS JEDLITSCHKA, ANDREA JANES, VALENTINA LENARDUZZI, AND XIAOZHOU LI, editors, *Product-Focused Software Process Improvement - 24th International Conference, PROFES 2023, Dornbirn, Austria, December 10-13, 2023, Proceedings, Part I*, **14483** of *Lecture Notes in Computer Science*, pages 363–378. Springer, 2023. 2

[5] SAMEER SHUKLA. **Streamlining integration testing with test containers: Addressing limitations and best practices for implementation**. *Inter. J. Latest Engg. Manag. Res.(IJLEMR)*, **9**:19–26, 2023. 2

[6] BRUNO NASCIMENTO, RUI SANTOS, JOÃO HENRIQUES, MARCO V. BERNARDO, AND FILIPE CALDEIRA. **Availability, Scalability, and Security in the Migration from Container-Based to Cloud-Native Applications**. *Comput.*, **13**(8):192, 2024. 2

[7] SARI SULTAN, IMTIAZ AHMAD, AND TASSOS DIMITRIOU. **Container Security: Issues, Challenges, and the Road Ahead**. *IEEE Access*, **7**:52976–52996, 2019. 2

[8] YUTIAN YANG, WENBO SHEN, BONAN RUAN, WENMAO LIU, AND KUI REN. **Security Challenges in the Container Cloud**. In *3rd IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications, TPS-ISA 2021, Atlanta, GA, USA, December 13-15, 2021*, pages 137–145. IEEE, 2021. 2

[9] QINGXIN XU, YU GAO, AND JUN WEI. **An Empirical Study on Kubernetes Operator Bugs**. In MARIA CHRISTAKIS AND MICHAEL PRADEL, editors, *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pages 1746–1758. ACM, 2024. 2, 53

[10] MARCO BARLETTA, MARCELLO CINQUE, CATELLO DI MARTINO, ZBIGNIEW T. KALBARCZYK, AND RAVISHANKAR K. IYER. **Mutiny! How Does Kubernetes Fail, and What Can We Do About It?** In *54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2024, Brisbane, Australia, June 24-27, 2024*, pages 1–14. IEEE, 2024. 2

[11] NIKOLAOS ASTYRAKAKIS, YANNIS NIKOLOUDAKIS, IOANNIS KEFALOUKOS, CHARALABOS SKIANIS, EVANGELOS PALLIS, AND EVANGELOS K. MARKAKIS. **Cloud-Native Application Validation & Stress Testing through a Framework for Auto-Cluster Deployment**. In *24th IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks, CAMAD 2019, Limassol, Cyprus, September 11-13, 2019*, pages 1–5. IEEE, 2019. 2, 53

[12] GIANLUCA TURIN, ANDREA BORGARELLI, SIMONE DONETTI, EINAR BROCH JOHNSEN, SILVIA LIZETH TAPIA TARIFA, AND FERRUCCIO DAMIANI. **A Formal Model of the Kubernetes Container Framework**. In TIZIANA MARGARIA AND BERNHARD STEFFEN, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*, **12476** of *Lecture Notes in Computer Science*, pages 558–577. Springer, 2020. 2

[13] JIE LU, CHEN LIU, LIAN LI, XIAOBING FENG, FENG TAN, JUN YANG, AND LIANG YOU. **CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis**. In TIM BRECHT AND CAREY WILLIAMSON, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 114–130. ACM, 2019. 2

[14] HAICHENG CHEN, WENSHENG DOU, DONG WANG, AND FENG QIN. **CoFI: Consistency-Guided Fault Injection for Cloud Systems**. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 536–547. IEEE, 2020. 2

[15] IVAN CILIC, PETAR KRIVIC, IVANA PODNAR ZARKO, AND MARIO KUSEK. **Performance Evaluation of Container Orchestration Tools in Edge Computing Environments**. *Sensors*, **23**(8):4008, 2023. 2

[16] B PURAHONG, J SITHIYOPASAKUL, P SITHIYOPASAKUL, A LASAKUL, AND C BENJANGKAPRASERT. **Automated Resource Management System Based upon Container Orchestration Tools Comparison**. *Journal of Advances in Information Technology*, **14**(3), 2023. 2

[17] TAO ZHENG, RUI TANG, XINGSHU CHEN, AND CHANGXIANG SHEN. **KubeFuzzer: Automating RESTful API Vulnerability Detection in Kubernetes.** *Computers, Materials & Continua*, **81**(1), 2024. 2

[18] RURIKO KUDO, HIROKUNI KITAHARA, KUGAMOORTHY GAJANANAN, AND YUJI WATANABE. **Application Integrity Protection on Kubernetes cluster based on Manifest Signature Verification**. *J. Inf. Process.*, **30**:626–635, 2022. 2

[19] GIORGIO DELL'IMMAGINE, JACOPO SOLDANI, AND ANTONIO BROGI. **KubeHound: Detecting Microservices' Security Smells in Kubernetes Deployments**. *Future Internet*, **15**(7):228, 2023. 2

[20] CLINTON CAO, AGATHE BLAISE, SICCO VERWER, AND FILIPPO REBECCHI. **Learning State Machines to Monitor and Detect Anomalies on a Kubernetes Cluster**. In *ARES 2022: The 17th International Conference on Availability, Reliability and Security, Vienna,Austria, August 23 - 26, 2022*, pages 117:1–117:9. ACM, 2022. 2

[21] ROEL J. WIERINGA. *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014. 17

[22] EARL T. BARR, MARK HARMAN, PHIL MCMINN, MUZAMMIL SHAHBAZ, AND SHIN YOO. **The Oracle Problem in Software Testing: A Survey**. *IEEE Trans. Software Eng.*, **41**(5):507–525, 2015. 28

[23] THOMAS BAILEY AND CRISTIAN CADAR. **Code, Test, and Coverage Evolution in Mature Software Systems: Changes Over the Past Decade**. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*, pages 210–220. IEEE, 2025. 29

[24] M S RAUNAK, D. R. KUHN, R. N. KACKER, AND Y. LEI. **Ensuring Reliability Through Combinatorial Coverage Measures**. *IEEE Reliability Magazine*, **1**(2):20–26, 2024. 29

[25] JIAWEI TYLER GU, XUDONG SUN, WENTAO ZHANG, YUXUAN JIANG, CHEN WANG, MANDANA VAZIRI, OWOLABI LEGUNSEN, AND TIANYIN XU. **Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management**. In JASON FLINN, MARGO I. SELTZER, PETER DRUSCHEL, ANTOINE KAUFMANN, AND JONATHAN MACE, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 96–112. ACM, 2023. 53, 54

[26] BINGZHE LIU, GANGMUK LIM, RYAN BECKETT, AND PHILIP BRIGHTEN GODFREY. **Kivi: Verification for Cluster Management**. In SAURABH BAGCHI AND YIYING ZHANG, editors, *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 509–527. USENIX Association, 2024. 53

[27] CECILIO CANNAVACCIUOLO AND LEONARDO MARIANI. **Automatic generation of smoke test suites for kubernetes**. In SUKYOUNG RYU AND YANNIS SMARAGDAKIS, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 769–772. ACM, 2022. 53

[28] CHRISTOPH REILE, MOHAK CHADHA, VALENTIN HAUNER, ANSHUL JINDAL, BENJAMIN HOFMANN, AND MICHAEL GERNDT. **Bunk8s: Enabling Easy Integration Testing of Microservices in Kubernetes**. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*, pages 459–463. IEEE, 2022. 53, 54

[29] DHANUSHKA JAYASURIYA. **Towards Automated Updates of Software Dependencies**. In ALEX POTANIN, editor, *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH Companion 2022, Auckland, New Zealand, December 5-10, 2022*, pages 29–33. ACM, 2022. 54

[30] STEPHEN MCCAMANT AND MICHAEL D. ERNST. **Early Identification of Incompatibilities in Multi-component Upgrades**. In MARTIN ODERSKY, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, **3086** of *Lecture Notes in Computer Science*, pages 440–464. Springer, 2004. 54

[31] ERIC RUIZ, SHAIKH MOSTAFA, AND XIAOYIN WANG. **Beyond api signatures: An empirical study on behavioral backward incompatibilities of java software libraries**. *Department of Computer Science, University of Texas at San Antonio, Tech. Rep*, 2015. 54

[32] ERIC HORTON AND CHRIS PARNIN. **V2: Fast Detection of Configuration Drift in Python**. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 477–488. IEEE, 2019. 54

[33] KURIENS SHAJI MALIEKAL. **AUTOMATED TESTING IN KUBERNETES: A COMPREHENSIVE FRAMEWORK FOR CODE QUALITY AND DEPLOYMENT ASSURANCE**. *International Research Journal of Modernization in Engineering Technology and Science*, 2025. 54

Appendix A

Reproducibility

A.1 Abstract

This artifact appendix describes the experiment set up as well as the implementation of the testing framework and the test suite.

A.2 Artifact check-list (meta-information)

- **Program:** CodeCanvas code base (private), Operator log processing script (`https://github.com/PinkRay7/operator_logs`)
- **Compilation:** Gradle (Kotlin)
- **Run-time environment:** Ubuntu 22.04.5 LTS, Kind, Kubectl (Client Version: v1.32.0, Kustomize Version: v5.5.0, Server Version: v1.33.1)
- **Hardware:** AWS Cloud
- **Execution:** TeamCity Build Agent
- **Metrics:** Failure detection rate, Operator command coverage
- **Output:** Execution logs and media (screen recordings and screenshots)
- **How much disk space required (approximately)?:** 32GB
- **How much time is needed to complete experiments (approximately)?:** For one target version per run, approximately 1h
- **Publicly available?:** No
- **Workflow framework used?:** TeamCity Kotlin DSL

A.2.1 How to access

not publicly available

A.3 Installation

JetBrains TeamCity, JetBrains Spacecode, kind cluster, docker (to publish CodeCanvas image)

A.4 Experiment workflow

As described in Section 6.1.

A.5 Methodology

Submission, reviewing and badging methodology:

- `https://www.acm.org/publications/policies/artifact-review-badging`
- `http://cTuning.org/ae/submission-20201122.html`
- `http://cTuning.org/ae/reviewing-20201122.html`