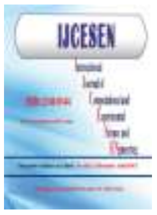
International Journal of Computational and Experimental Science and ENgineering
(IJCESEN)

Vol. 11-No.3 (2025) pp. 5422-5441
http://www.ijcesen.com

Copyright © IJCESEN



ISSN: 2149-9144

Research Article

Development of a Simulator to Mimic VMware vCloud Director (VCD) API Calls for Cloud Orchestration Testing

Zahir Sayyed[*]

R&D Engineer Software, Jamesburg, New Jersey, USA
*** Corresponding Author Email**: sayyedzahirr@gmail.com - **ORCID**: 0009-0004-1148-4126

Article Info:

DOI: 10.22399/ijcesen.3480
Received : 14 May 2025
Accepted : 16 July 2025

Keywords

Cloud Orchestration,
vCloud Director,
API Simulation,
DevOps Testing,
CI/CD Automation

Abstract:

Orchestration systems, particularly those on VMware vCloud Director (VCD), play a vital role in managing multi-tenant virtualized environments. Nonetheless, it is problematic to test automation scripts and orchestration workflows directly on production or staging VCD infrastructure: it is expensive, inaccessible, and may endanger live services. This paper provides an overview of the design and implementation of an API call simulator tailored to a specific domain, aiming to create safe, efficient, and repeatable testing environments for developers and DevOps engineers. In contrast to generic mocking tools, this simulator offers a feature set tailored to VCD-specific requirements, including stateful API behavior, vApp mock lifecycles, and dynamic responses. It confirms popular HTTP requests on core end-points such as sessions, vApps, catalogs, and networks, offering a precise test proxy that does not map virtualization to the backend. The simulator also fits well in CI/CD environments and facilitates chaos testing through fault injection. A detailed analysis demonstrates its high fidelity to real VCD behavior, with low latency under concurrent load, and developers were satisfied with the results. Applications include use as a development sandbox tool, a disaster recovery testing tool, an educational tool, and a certification tool. The paper concludes by suggesting the adoption of these approaches on a broader scale, both in enterprise settings and those involving cloud training. The scalability of the simulator ultimately addresses the continuity limitations of present-day testing in cloud orchestration.

1. Introduction

Automation and orchestration of complex infrastructure are central concepts in the context of modern cloud computing, particularly within a multi-tenant environment. VMware vCloud Director (VCD) is one of the most popular platforms in the business. Through VCD, cloud providers can deploy secure, multi-tenant virtual data centers to customers, enabling them to centrally manage and scale other resources, including virtual machines, networks, and storage. VCD is strong, though; however, it is not very user-friendly to verify through a live setup. Developers and DevOps engineers may encounter difficulties when required to run orchestration scripts or automation jobs that interact with the VCD API. The fundamental issue is that testing in a real VCD environment is hazardous and cost-prohibitive. A test that calls on the production VCD server can cause service disruption, raise security issues, or cause live infrastructure damage in the event of a failure. Moreover, not all teams have the opportunity to run a dedicated VCD testbed due to licensing and operational costs.

This article focuses on the design and creation of a simulator that imitates the actions of VMware vCloud Director API calls. The simulator is designed to enable developers and testers to execute orchestration workflows realistically without requiring a connection to a live VCD environment. Like the actual VCD API, the simulator implements standard RESTful requests (GET, POST, PUT, and DELETE) and responds accordingly. Considering the example of a test script attempting to create a virtual application (vApp), the simulator returns structured data that appears identical to what VCD would have, allowing the script to continue running and being checked without using any actual infrastructure.

There are several benefits to this strategy. First, it becomes safer and easier to test the complicated workflows, and there is no chance of confusing the actual services. Second, it reduces the development and quality assurance costs, as there is no need to have a complete cloud infrastructure that must be provisioned and maintained in the test environment. Third, it can improve iteration and automation speed in continuous integration and delivery (CI/CD) pipelines, as test speed and reliability are essential factors.

This article aims to achieve three objectives. To begin with, it proposes presenting a realistic approach to implementing a simulation of the VCD API using modern web technologies. The second point clarifies that the simulator can be set up to provide various kinds of responses, allowing users to understand how their automation reacts to positive and negative situations, as well as edge cases. Third, it assesses how this simulator facilitates development processes, particularly when using automated tests. There is a need to explain the extent of this work. The simulator does not attempt to mimic the complete operations of the VCD system, including real provisioning of virtual machines or hypervisor-level control. Instead, it only emphasizes responses at the API level. This makes it perfect for functional testing, regression testing, and integration testing, where the primary focus is whether script and software tools are interacting appropriately with the API endpoints, as opposed to what is happening in the background of the virtual infrastructure.

2. Background and Problem Statement

2.1 Overview of VMware vCloud Director (VCD)

VMware vCloud Director (VCD) is a cloud service delivery platform that service providers use to provision and orchestrate Infrastructure as a Service (IaaS). It is built on top of the core VMware virtualization infrastructure—namely vSphere and NSX—and reflects a multi-tenant abstraction layer. This design enables cloud providers to serve independent customers through the creation of virtual data centers (VDCs), where compute, network, and storage resources are securely isolated and managed. Such a layered orchestration model aligns with broader strategies in infrastructure management, including dual sourcing and resource decoupling, which improve system resilience and vendor flexibility (16). In essence, VCD provides a RESTful application programming interface that governs the allocation of virtual resources by orchestrating the process. Such resources are virtual machines (VMs), vApps (i.e., one or more VMs that comprise virtual applications), catalogs (which hold templates and media), and networks. The API can automate nearly all tasks that can be driven through the web-based user interface, including creating a vApp, uploading ISO images, managing virtual networks, and configuring firewall rules.

The VCD API is organized and operates according to standard guidelines (24). For example, a user can log in using a session endpoint, retrieve a list of available catalogs, or initiate the creation of a virtual machine with specific parameters. This makes the API a critical inclusion in any DevOps or automation process in VCD environments. These API calls are frequently used by teams in the form of scripts and in CI/CD pipelines, where they automate operations such as environment creation, testing, and destruction. As the figure below illustrates, VCD operates atop VMware Cloud Foundation, leveraging the underlying infrastructure to deliver flexible, scalable, and secure cloud environments. The integration of API-driven orchestration into developer pipelines allows for dynamic environment creation, automated testing, and seamless teardown—essential for agile and iterative delivery models.

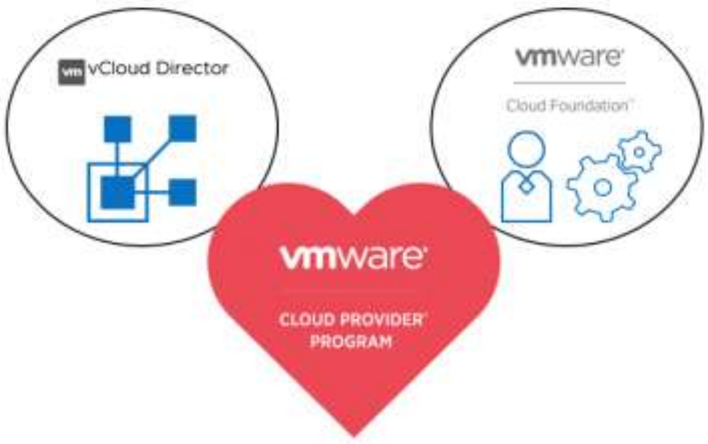


Figure 1: *VMware vCloud Director + VMware Cloud Foundation = Harmony*

2.2 Issues with Direct Testing of VCD Environments

Although the VCD API has powerful capabilities, direct testing against the API poses several significant problems (14). The high cost of infrastructure is a considerable concern. Environments based on VMware are very costly to license and to maintain. A basic VCD environment, including all its dependencies such as vCenter, ESXi hosts, NSX networking, and external databases, consumes a considerable number of resources. This renders it uneconomical for several organizations to maintain a unique test environment for API validation. The next significant problem is access restrictions. Security, compliance, and stability are essential factors in preventing easy access to the production VCD environment in many organizations. Often, developers and testers lack the necessary permissions to conduct meaningful tests or execute orchestration scripts. Even when access is granted, calling experimental APIs can be risky. For instance, an inefficiently written script may inadvertently shut down a critical virtual machine, delete sensitive data, or alter network configurations—disrupting live services in ways that compromise both availability and reliability. These risks are particularly acute in systems that require strong data consistency and operational resilience, such as those underpinned by cloud-native databases like MongoDB (11). Furthermore, attempts to scale or test these systems in real time without controlled environments can introduce performance bottlenecks or data integrity issues, reinforcing the need for isolated simulation layers (12).

Production testing also raises the issue of testing stability and performance. Running repeated tests at ridiculous levels by creating, modifying, and deleting virtual resources may unnecessarily burden the production system. This may cause a delay in real operator performances or alarm the supervisory systems. Additionally, in many production settings, failure conditions analogous to those that testers should simulate (e.g., timeouts, failing responses, or network suspensions) are not generated, which prevents tests from validating how orchestration scripts react to them. These limitations typically leave teams with two unfavorable options: either perform against a real system and take the risk, or perform against nothing and take a chance when the scripts are deployed. Both approaches are less than ideal and can lead to time wastage, unsuccessful deployments, and the inability to identify bugs at the earlier stages of development.

Testing against a live VMware vCloud Director (VCD) environment introduces numerous challenges that hinder safe, efficient, and repeatable testing. As shown in Table 1, issues such as high infrastructure costs, limited access rights, and the risk of unintentional disruptions make real-world VCD environments unsuitable for development pipelines.

Table 1: *Challenges of Direct VCD API Testing*

Challenge Area	**Details**
Infrastructure Cost	High licensing and maintenance costs for VCD, vCenter, ESXi, NSX, and databases.
Access Restrictions	Limited developer/tester permissions; compliance and stability concerns restrict meaningful testing.
Risk of Disruption	Experimental scripts can accidentally shut down VMs, delete data, or alter network settings.
Production Load & Stability	Repeated test cycles can slow down or destabilize production environments.
Lack of Simulated Failures	Difficult to test failure conditions (e.g., timeouts, network issues) in live environments.
Risk vs. Blind Testing	Teams must choose between risky real-environment tests or no testing at all before deployment.
Development Impact	Leads to time loss, failed deployments, and bugs discovered late in the cycle.

2.3 Testing Challenges in DevOps Contexts

Orchestration systems, particularly those on VMware vCloud Director (VCD), play a crucial role in managing multi-tenant virtualized environments. However, running automation scripts and orchestration workflows directly against production or even staging VCD infrastructure can be highly problematic due to operational costs, restricted accessibility, and risks to live services. These challenges necessitate robust simulation alternatives that allow for safe and efficient testing. This paper provides an overview of the design and implementation of an API call simulator tailored to the unique demands of cloud orchestration environments—offering developers and DevOps engineers a secure, repeatable environment for experimentation and validation. Such approaches align closely with DevSecOps principles, where secure and automated testing is integrated throughout CI/CD pipelines to minimize production risks and improve code reliability (20). In contrast to generic mocking tools, this simulator offers a feature set tailored to VCD-specific requirements, including stateful API behavior, vApp mock lifecycles, and dynamic responses (32). It supports the prevalent HTTP operations on primary endpoints, such as sessions, vApps, catalogs, and networks, providing a precise testing substitute that does not require any form of virtualization on its backend. The simulator also fits well in CI/CD environments and facilitates chaos testing through fault injection.

A detailed analysis demonstrates its high fidelity to real VCD behavior, with low latency under concurrent load, and developers were satisfied with the results. Sandbox applications and disaster recovery testing are possible uses, as well as applications in learning and certification. The paper concludes by offering suggestions for adopting them on a broader scale, both in enterprise settings and those involving cloud training. The simulator is an ultimate scalable remedy to contemporary testing limitations in cloud orchestration.

3. Literature Review and Related Work

3.1 Existing Mocking and Simulation Tools

Mocking software is increasingly common in software development to simulate the responses of real systems in a comprehensible and repeatable manner (33). These tools enable developers to decouple system components, emulate external service behavior, and run integration tests without depending on live systems. Among the most widely used mocking tools are WireMock, Postman Mock Server, and Beeceptor. Each offers practical capabilities for simulating HTTP-based APIs and is effective in many general-purpose software projects. However, none of these tools is explicitly designed to simulate the complex workflows or stateful interactions required in cloud orchestration platforms like VMware vCloud Director (VCD). In specialized domains—whether in healthcare systems, financial platforms, or virtualized infrastructure management—general-purpose mocking often fails to account for chained state transitions, conditional logic, or role-sensitive behavior (28). Therefore, while useful in early development stages or lightweight service mocking, such tools lack the depth and domain-awareness needed for realistic cloud orchestration testing.

An example is WireMock, an open-source mechanism that enables the definition of HTTP stubs with predetermined responses. It allows flexible matching rules, is responsive to delay, and may provide dynamic content in response to request parameters. It is also quite configurable and can be well used to test microservices. WireMock does not, however, provide out-of-the-box support for modeling complex state transitions or supporting long-lived workflows found in cloud orchestration systems. It is very good at mocking out a single request-response pair, but fails to simulate the rest of the lifecycle when working with a virtual application or the chain reactions that occur when a resource is provisioned in a cloud-based scenario.

Another common alternative is Postman Mock Server, specifically used more often to test frontend and API integration. It enables teams to create API page designs as well as fake endpoints that respond with a set JSON reply. This comes in handy at an early developmental stage when the backend service might not be ready. Although Postman accepts environment variables and dynamic samples, it continues to work primarily in the arena of static response generation. It lacks memory or state persistence between calls, which is necessary when attempting to emulate a real-world application where some sequence of API calls results in a change in system state. Beeceptor is a straightforward, cloud-based, HTTP mocking server that specializes in capturing and debugging RESTful API traffic. Installation is straightforward, and it offers a user-friendly interface for adding and checking traffic rules. It is usually applied in prototyping and simulating third-party services. It, however, lacks support for more advanced functionalities such as request chaining, in-memory state monitoring, or emulation of authentication flows typical of VCD platforms. Each of these tools has a vital purpose and can be helpful in numerous testing situations. Nevertheless, they have not been constructed from a cloud orchestration perspective (3). They specialize in individual requests instead of workflows, where tasks are tied to one another, which is typical of cloud infrastructure management. The state of a particular API call sometimes dictates how subsequent APIs can and do act, and generic mocking tools cannot readily support this type of interaction.

As the figure below illustrates, general-purpose tools are typically request-focused, whereas orchestration testing demands a flow-focused, context-aware simulation approach.

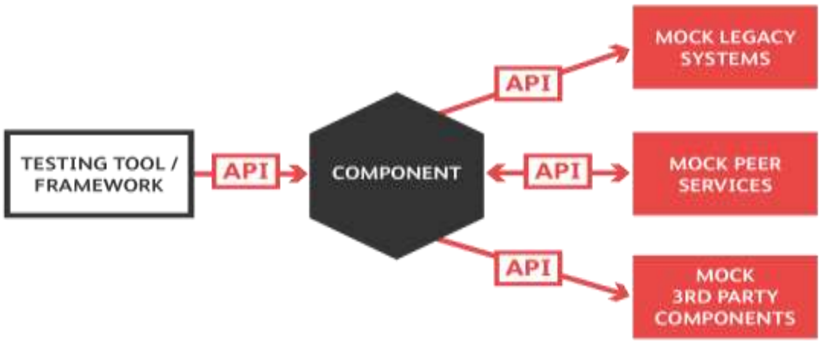


Figure 2: *Mock Testing*

3.2 Gaps in Current Solutions

Although available tools offer valuable components on how to mock APIs, they lack the components needed to mock VMware vCloud Director Environments. A crucial shortcoming is the lack of representation for stateful operations. Operations such as establishing a vApp, turning it on, or attaching a network are not simple calls in VCD, but rather part of a larger lifecycle that spans several stages and system elements. A simulator that provides only fixed replies is not able to truly represent VCD behavior when time-marched. Support for cloud-native data structures and payload formats is another major shortcoming. VMware vCloud Director (VCD) relies heavily on complex XML and JSON schemas to represent its resources—ranging from virtual machines and media files to catalogs and virtual networks. These schemas are often deeply nested and interlinked, referencing other entities to define dependencies, lifecycles, and orchestration behavior. Generic mocking frameworks, however, typically treat payloads as opaque blobs without enforcing schema validity or mimicking inter-resource linkage. This limitation reduces test realism and may lead to integration failures when such assumptions are carried into production workflows. Modern event-driven and distributed systems demand fault tolerance and context-aware message structures, where improper schema handling can compromise system behavior (7). Identifying and respecting context boundaries is essential when decomposing monolithic systems into modular services—underscoring the importance of accurate, schema-compliant communication between components, something generic mock servers often fail to reproduce (8).
Authentication and authorization are also weakly supported. VCD uses a session token and role-based access control to control permissions between tenants and users. Available tools simulate basic auth or API keys and do not attempt to emulate session management, token expiry, isolation, or multi-tenant isolation. This complicates testing security-sensitive operations or verifying that scripts behave reasonably when subjected to the actual access limitation. Additionally, many of these tools lack the development of failure conditions and error simulation (5). In practical VCD settings, API call failures often occur due to resource constraints, rights concerns, and internal system issues. The ability to emulate such failures is crucial for creating robust and resilient orchestration systems. It is also challenging to test edge cases without the capability to simulate faults and timeouts, or even inject errors and timeouts.

3.3 Need for Domain-Specific Simulators

The above challenges indicate the need for a domain-specific solution that suits the special purpose of VMware vCloud Director. Such a simulator needs to have an idea of the domain model of VCD, its resources, lifecycles, workflows, and authentication methods, unlike a general-purpose API-mocking tool. It should be able to replicate real-life behavior in a stateful, predictable manner, allowing for realistic testing of orchestration scripts and automation pipelines. By designing a VCD-specific simulator, the reliability of testing can be significantly enhanced, infrastructure costs can be reduced, and the lifecycle can be shortened (2). It enables testers to simulate real-life experiences without compromising live systems. Preserving internal state allows it to simulate entire vApp lifecycles, catalog management, and user sessions, and in doing so, offer more valuable results than simple, static mock servers.
This also aids in simulating error cases and edge cases, allowing developers to test their system in both typical and failure scenarios. In addition to development and testing, such a simulator can also be beneficial in educational and training settings. It can provide students or junior engineers with a safe sandbox to learn the functions of VCD, call API functions, and conceptualize orchestration flows, without needing to use a live environment, which is both expensive and complicated. Although broader-scale mocking tools are helpful during numerous

software projects, they cannot fully simulate VMware vCloud Director Functionality. The available tools leave gaps that need to be filled by a domain-specific simulator, as well as gaps in terms of stringent testing and development requirements posed by today's cloud orchestration workflows. To overcome the challenges identified in live VCD testing (as shown in Table 1), a tailored simulation solution is necessary. As outlined in Table 2, a VCD-specific simulator brings domain awareness, cost efficiency, and error simulation capabilities that generic mocking tools lack.

Table 2: *Justification for a VCD-Specific Simulator*

Requirement/Benefit	**Explanation**
Domain Awareness	Simulator should understand VCD's specific resources, workflows, and authentication methods.
Stateful Behavior	Must replicate real-world state changes (e.g., vApp lifecycles, user sessions) for accurate testing.
Enhanced Reliability	Realistic simulations improve confidence in automation scripts and reduce test ambiguity.
Cost Efficiency	Reduces need for costly VCD environments; testing becomes feasible for smaller teams or projects.
Error & Edge Case Simulation	Enables safe, repeatable testing of failure conditions not easily recreated in production.
Educational Utility	Acts as a training tool for students and junior developers without needing access to live systems.
Superior to General Mocking Tools	Fills gaps left by generic API mocks which lack VCD-specific features and behavior.

4. Architecture of the Simulator

4.1 Core Components

The simulator that models VMware vCloud Director (VCD) API calls follows a modular design that intentionally echoes the organization and behavior of genuine VCD sessions. Its framework consists of several interdependent modules, and each module contributes to a believable and repeatable testing environment (26). Central to the simulator is the API Endpoint Manager. It listens for all incoming HTTP requests from client software, including orchestration scripts and automation utilities. After receiving a message, the manager determines which endpoint the client targeted, verifies the HTTP method used (GET, POST, PUT, or DELETE), and forwards the request to the appropriate internal handler. In this way, the manager serves as the gateway that mediates between external callers and the simulators' underlying processing logic.

Once a request has been routed, control passes to the Payload Engine, which constructs the response payload destined for the client. This engine operates from predefined templates that closely mirror the structure and content of genuine VCD API outputs, inserting dynamic elements such as unique identifiers, timestamps, and operational status according to the current state of the simulation. Such contextual variability is crucial for replicating production-like workflows, where the content of a response is contingent upon prior actions and current parameters. To further sustain the appearance of an operational cloud environment, the simulator employs a State Tracker (23). This component monitors all active entities within the simulation—virtual applications (vApps), networks, catalogs, and user accounts—and updates its internal registry whenever a client request creates, modifies, or removes a resource. By recording every transaction, the State Tracker ensures that future queries reference an accurate snapshot of the system's current state. For instance, when a client creates a vApp and subsequently requests a list of vApps, the simulator dutifully includes the new entry in the returned dataset, even though no underlying infrastructure has been provisioned.

The final critical element is the Logging Module. It captures each incoming request, together with the corresponding response, headers, payload, and a precise timestamp. Such detailed records aid in debugging, facilitate systematic audits, and clarify the execution flow of test scripts during simulated runs. Furthermore, they enhance transparency and, more importantly, provide the traceability that regulated testing scenarios demand. When coupled, these four parts operate seamlessly to produce a simulation platform that is flexible, responsive, and mindful of its internal state, thereby mirroring VCD's API layer with high fidelity.

As the figure below illustrates, the simulation framework mirrors the operational logic and interaction flow of a real-world VCD API layer.

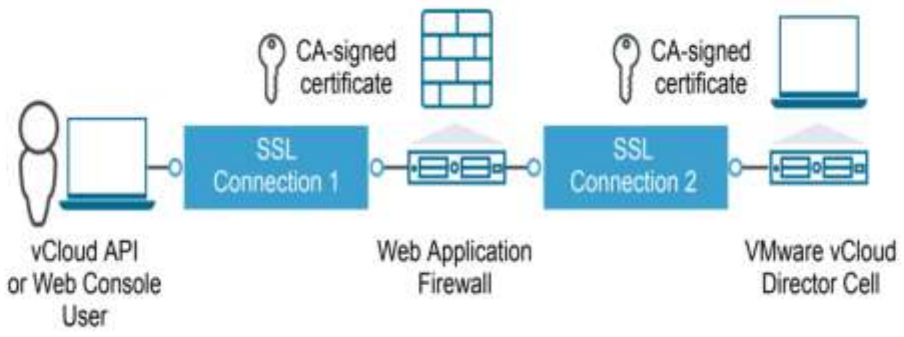


Figure 3: *VMware vCloud Director*

4.2 Technology Stack

The simulator is constructed from a blend of contemporary, broadly supported technologies selected for their ease of use, solid performance, and room for future growth. Backend logic runs on either Node.js or Flask, with the choice guided by deployment context and performance criteria. Both environments come with mature libraries for RESTful routing, middleware insertion, and request parsing, making either one a suitable base for a system that must juggle multiple endpoints, each capable of shifting behavior. The API's structure is defined using the OpenAPI Specification, also known as Swagger (6). By formalizing every path, parameter, response, and possible error, this contract ensures all endpoints remain consistent and predictable. Automated tools then read the contract to build live documentation and create tests, so developers can immediately see how to use the API in systems such as Postman or Swagger UI while they are still working on other test scenarios.

The simulator manages its data using a lightweight, schema-less database, typically either MongoDB or Redis. MongoDB's document model suits VCDs' nested JSON structures, allowing developers to store entire payloads as single, retrievable documents. Redis, however, shines when speed is critical; its in-memory key-value store can quickly track session states or ephemeral resources needed during high-frequency CI/CD cycles. Combining Node.js or Flask with the OpenAPI spec and MongoDB or Redis yields a nimble stack that emphasizes speed, realism, and the flexibility for users to adjust parts of the system without waiting for slow, monolithic releases.

4.3 Security Simulation Features

To function as an accurate stand-in for an actual VCD environment, the simulator must replicate VCD's authentication and security guardrails. The first element of this emulation is token-based session management. When a client directs a login request to the simulator's authentication endpoint, the simulator issues a token, usually structured as a JSON Web Token (JWT) or a mock bearer token. That token must accompany subsequent API calls, mirroring the behavior of a live session. In each request, the simulator verifies the token and can mark it expired either after a predetermined duration or according to configurable session rules. In addition to the token flow, the simulator offers a basic-auth emulation for users who expect a more classical scheme. Under this method, the system checks supplied username and password pairs against a static inventory defined in its configuration file. Although real user accounts are neither created nor persisted, the simulator upholds access-control rules and replies with standard HTTP status codes whenever authentication fails or succeeds.

To facilitate deployment in secure test labs, the simulator optionally accepts Secure Socket Layer (SSL) termination (27). When configured in this manner, the tool presents an HTTPS endpoint, enabling client applications to exchange data with it over an encrypted channel, just as they would with a production cloud service. This feature meets the stringent security policies often found in enterprise networks, where plaintext traffic is explicitly forbidden even during non-production testing (21). With realistic security implemented, the simulator can be integrated into pipelines that rely on authentication tokens, session cookies, and protected transport, thereby strengthening its credibility and broadening its applicability in cloud-orchestration evaluation.

5. Core Features and Functionalities

The simulator, which mimics VMware vCloud Director (VCD) API calls, has been designed with a set of core features that replicate authentic API behavior within a controlled, adjustable environment. These features answer the everyday demands of testing cloud orchestration, validating automation procedures, and aiding development work. Each function reacts in a manner that remains true to the responses an actual VCD system would issue, ensuring that the test results remain meaningful and dependable.

As the figure below illustrates, these functionalities form the operational heart of the simulator, enabling realistic cloud behavior emulation without the cost, risk, or complexity of live VMware infrastructure.

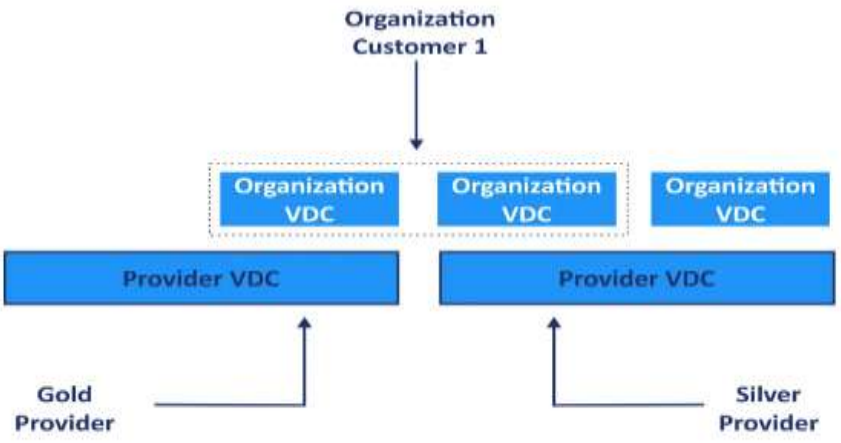


Figure 4: *VMware vCloud Director*

5.1 Simulated Endpoint Catalog

Central to the simulator is a comprehensive catalog of mock API endpoints that closely match the key tasks typically carried out in VCD settings. These endpoints follow the same structure and functions as seen in a live VCD API, covering familiar entry points, including the authentication call that initiates a session, as well as those for managing virtual applications (vApps), catalogs, media objects, and network settings. For instance, when a client issues a login request to the session endpoint, the simulator creates a session token and returns it in the same format that production VCD instances use (17). Once the client is authenticated, it can then call endpoints to create a vApp, list catalog media items, or touch the virtual network layer. Each endpoint emulates the behavior outlined in the official VCD API documentation, delivering consistent response structures, status codes, and headers. Because it exposes this wide set of endpoints, the simulator can act as a drop-in substitute for live VCD systems in many test cases where the tester wants to validate automation scripts or orchestration logic without consuming real cloud resources.

5.2 Dynamic Response Generation

Another key capability of the simulator lies in its ability to craft dynamic responses influenced by incoming request data, environment settings, and its simulated state. Instead of relying on static files or hardwired replies, the simulator utilizes structured templates—typically in JSON—that contain variables and placeholders. When a request arrives, the system locates the matching template and populates its fields using values extracted from the request, pulled from environment variables, or gathered from internal state. The described method generates context-aware replies that closely mirror behavior seen in production systems. For example, when a user submits a command to provision a new vApp, the framework automatically fabricates a fresh identifier, records a timestamped deployment status, and formats a resource URL in line with documented conventions. All of these elements are created on the fly and woven into the reply template during the actual execution cycle (25). Generating responses in this manner is beneficial for workloads that require strict structural consistency or for test scenarios where analysts want to observe how automation scripts behave when presented with subtly different data. By injecting controlled variability yet still conforming to well-defined patterns, the simulator broadens test scope without sacrificing the reliability of individual cases.

5.3 Stateful Operation Handling

Preserving system state across multiple chained API calls presents a persistent hurdle in replicating sophisticated orchestration backends, such as VMware Cloud Director. The current implementation addresses this problem by incorporating an internal state table that logs each client action, enabling returned messages to reflect the evolving context accurately. Preserving state across client transactions is essential for accurately imitating the lifecycle of virtual applications. When a request arrives to provision a new vApp, the simulator commits the vApp's name, status, and configuration to either in-memory storage or a persistent backend. Subsequent commands, whether to power the vApp on or to check its state, are processed using this retained information. As a result, the simulator models the temporal evolution of VCD resources and guarantees that orchestrated workflows behave consistently with production behavior. Having a coherent state across sessions also underpins multi-step testing, where the output of one API call determines the validity of later calls. This continuity reflects genuine usage patterns, allowing developers to exercise the entire orchestration pipeline and confirm that intermediate states remain valid.

5.4 Fault Injection Mechanism

To benchmark robustness, the simulator incorporates a fault injection feature that lets testers introduce delays, errors, or anomalous behavior on demand. By engineering scenarios such as network timeouts, downstream service outages, or corrupt payloads, users can verify whether calling clients handle adversity gracefully, all without impacting production systems. For instance, a tester can set the simulator to pause for a few seconds before responding. This delay accurately imitates the noticeable lag that users might experience when systems are under heavy load. At other times, the simulator can emit specific HTTP error codes—such as 503(Service Unavailable) or 500 (Internal Server Error)—accompanied by clear text messages that guide the client about the nature of the fault.

By injecting these conditions, developers can observe how their automation behaves during failure scenarios, verifying that fall-back routines, retry loops, and error logs operate as intended. Such testing is invaluable in continuous integration pipelines and in mission-critical scripts, where uninterrupted operation and proven fault tolerance are non-negotiable. Taken together, the simulator's precise endpoint imitation, dynamic response settings, stateful behavior modeling, and customizable fault generators create a solid environment for validating cloud orchestrators. Developers and testers can explore intricate, real-world interactions with the VCD API without endangering production resources, incurring unexpected costs, or managing the overhead of live infrastructure. As a result, the tool has become a standard component of contemporary DevOps practices and cloud-centered development workflows.

6. Methodology

The simulator, designed to reproduce VMware vCloud Director API exchanges, evolved through a methodical, multi-phase process (15). Each discrete phase was oriented toward a specific technical goal, thereby demonstrating anticipation of real-world testing and automation environments in which the simulator might be deployed. The sequence of activities included schema extraction, stub generation, middleware assembly, pipeline integration, and the judicious application of contemporary development tools, all aimed at delivering a durable and versatile simulation platform.

6.1 API Schema Extraction and Stub Generation

The inaugural task consisted of harvesting the authoritative API schema records from VMware's vCloud Director Documentation set. These records enumerate accessible endpoints, define request formatting, clarify anticipated responses, list error codes, and outline the requisite authentication flows. To capture this information in a coherent and computable format, the team adopted the OpenAPI (formerly known as Swagger) specification, thus facilitating future validation, code generation, and automated testing activities. Once the API schema was finalized, the team turned its attention to creating preliminary endpoint stubs, employing a utility such as Swagger Codegen. This utility ingests an OpenAPI specification and produces skeletal server code in multiple languages, relieving developers of much of the boilerplate typically involved in endpoint definition. The resulting stubs include route bindings, supported HTTP verbs, basic request validation, and empty handlers intended for future logic. By providing this initial scaffold, the team was able to focus sooner on the substantive features of the simulator rather than repetitive configuration tasks. These automatically produced files became the baseline for the simulator's routing component and were later adjusted to accommodate stateful sessions and context-sensitive replies. Customizing the scaffolding at this stage helped keep the simulator's API structure consistent with the production VCD service and verified that all routing rules were present before more complex functionality was layered on top.

6.2 Middleware Development

With the endpoint scaffolds in place, attention shifted to the middle tier that orchestrates request handling, embeds application logic, and generates dynamic outcomes in response to client messages. At this stage, the team implemented stateful tracking as its cornerstone feature. Dedicated middleware modules now log the status of every entity—vApps, networks, catalogs, and media items—as it is created, modified, or deleted via any simulator endpoint. Because of this systematic logging, the simulator can reproduce lifecycles with a high degree of verisimilitude; for instance, a vApp moves from "creating" to "ready" automatically, and queries for media items return precise, up-to-date lists. Developers built a payload resolution engine to facilitate the generation of dynamic responses (22). This engine reads pre-defined templates and substitute's placeholders with values drawn from the incoming request, the environment configuration, or the live system state. Thus, a single template might receive a newly assigned identifier, user-provided metadata, or the current timestamp, allowing the simulator to adapt quickly to a variety of test conditions. Together, these middleware components enable the system to mimic real-world orchestration flows in a repeatable and dependable way, without depending on actual hardware or external services.

As the figure below illustrates, middleware acts as the connective tissue of the simulator, coordinating between endpoint input and internal logic to generate intelligent, context-sensitive responses.

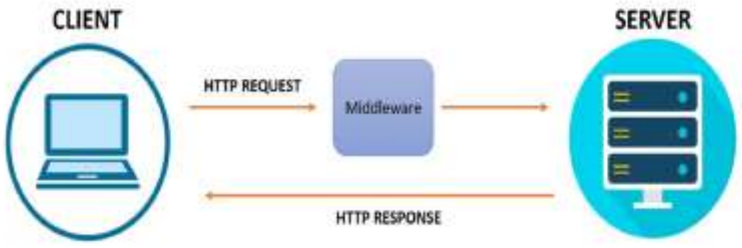


Figure 5: *Middleware*

6.3 Testing and CI/CD Pipeline Integration

To facilitate use in contemporary development workflows, the simulator was configured for straightforward deployment within CI/CD pipelines. The initial step in this approach involved containerizing the application with Docker. A dedicated container was constructed to bundle the simulator, its dependencies, configuration files, and the complete runtime environment in a single image. As a result, users can execute the simulator on any machine with Docker installed, eliminating the need for manual setup replication. Following containerization, the simulator was integrated into CI/CD platforms, including GitHub Actions and Jenkins. These systems trigger automated testing and deployment sequences with every code commit or build event. By inserting the simulator into the pipeline as a discrete stage, orchestration scripts and associated automation logic can be exercised against a stable API mock during each build. This practice enhances test reliability and reduces the likelihood of surface-level integration failures surfacing only at later stages of the release cycle (29, 30). Pipeline integration equally accommodates the running of custom test suites, preliminary environment scripts, and cleanup routines. In this role, the simulator becomes a key fixture of the automated threshold, supporting quick, repeatable, and secure testing sequences.

6.4 Tooling Stack

The final methodological phase focused on selecting and integrating tools to cover development, simulation, testing, and deployment. The team documented the API structure using OpenAPI, and throughout the entire lifecycle, Swagger UI provided interactive references, and Swagger Codegen generated initial stub files. To ease portability, the application was containerized with Docker, allowing the simulator to launch on developer laptops, virtual machines, or cloud services with little extra configuration. Testing teams relied on Postman to design and execute collections against the simulator, confirming that actual responses matched expected behavior across diverse scenarios.

Continuous integration pipelines in GitHub Actions and Jenkins executed automated tests, built fresh container images, and published them to shared registries for joint exploration. These routines ensured that the simulator remained in working order whenever code was added or altered, supporting quick feedback for all contributors. Following an orderly sequence—schema design, logic coding, container packaging, and automated release—the project produced a realistic, efficient tool for evaluating cloud orchestration where hands-on access to VMware vCloud Director is impractical.

As shown in Table 3, GitHub Actions and Jenkins were used to automate testing, containerization, and release workflows. This not only accelerated development cycles but also guaranteed that code changes did not compromise the simulator's core functionality.

Table 3: *CI/CD Pipeline and Simulator Development Workflow*

Stage / Component	**Details**
CI Tools Used	GitHub Actions and Jenkins
Automation Functions	Ran automated tests, built new container images, and published to shared registries
Purpose of CI	Ensured simulator functionality was intact after code changes; enabled fast feedback loops
Development Sequence	Schema design → Logic coding → Container packaging → Automated release
Outcome	Delivered a realistic, efficient tool for testing VCD orchestration workflows without direct access

7. Practical Use Cases and Applications

The simulator for VMware vCloud Director (VCD) API calls was engineered as a hands-on utility rather than an abstract exercise, addressing the everyday problems that operators and developers encounter. Its thoughtful architecture and broad feature set allow it to fit smoothly into contemporary DevOps pipelines, cloud application projects, and classroom demonstrations. By acting as a safe and budget-friendly substitute for an active VCD instance, the simulator adds considerable utility where security policies, cost controls, or the need for predictable, repeatable tests take priority.

7.1 CI/CD Pipeline Testing

Among the many scenarios where the simulator shines, integration and testing within continuous integration and continuous delivery (CI/CD) pipelines is the most immediately valuable (34). In typical cloud-native workflows, automation scripts touch VCD APIs to carry out tasks such as provisioning VMs, configuring networks, or rolling out updated applications. Verifying these scripts early and often within the pipeline is critical; however, routing every build through a production-like VCD environment is neither cost-effective nor sufficiently secure. Integrating the simulator directly into the continuous integration pipeline enables developers to verify orchestration logic with every code commit or build. Acting as a mock VCD platform, the simulator predictably responds to API calls, eliminating ambiguity during test execution. Because of this stable feedback, integration failures, schema mismatches, and logic bugs can be detected before they reach production. The setup also allows multiple pipeline runs to proceed in parallel, sidestepping concerns about infrastructure contention or external dependencies (31).
Packaged as a container, the simulator starts and stops almost instantly with each pipeline execution, leaving no residual state behind. This speed ensures that every test runs in a clean, repeatable environment, eliminating the need for build engineers to spend precious minutes troubleshooting artifacts from prior runs. The result is a marked boost in testing reliability and a better fit with the rapid iteration cycles modern teams have come to expect.

7.2 Developer Sandboxes

When cloud engineers and developers write automation or orchestration scripts, they need a fast, forgiving space to experiment without risk. Targeting a live VCD system directly exposes the project to accidental changes on shared resources, unwanted service charges, and possible downtime. Worse still, the pressure of working on production infrastructure often forces developers to limit their experiments, stifling innovation and learning. A purpose-built sandbox eliminates these dangers, encouraging developers to test boldly and iterate quickly while preserving the stability of the broader system.
The simulator addresses these needs by offering a controlled sandbox that closely duplicates VCD's behavior while leaving the production stack untouched. Within this isolated space, engineers can easily load new scripts, check the sequence of API calls, and observe how their logic responds to different inputs. They can repeat those tests at will, because every run starts from the same known state. Because the simulator tracks state and sends dynamic replies, developers can push through entire workflows-authentication, resource provisioning, and final teardown-and see precisely how their code would behave against the live API. As a result, teams shorten the time between draft and deployment, enter production with more substantial confidence, and encounter fewer unexpected errors after integration.

7.3 Disaster Recovery and Failover Simulations

The simulator also proven invaluable for planning disaster-recovery (DR) and failover strategies. Real-world DR exercises must verify intricate failover paths, confirm that error-handling code operates as intended, and mimic outages at both component and region levels. Testing those scenarios on a live system is disruptive, expensive, and often prohibited in environments where every minute of downtime incurs a significant cost.

The simulator creates a safe and repeatable testing space in which disaster-recovery scenarios can be executed without impacting production services. With its fault-injection options, users can simulate various errors—such as API timeouts, transient service outages, and even malformed responses—and then observe how the system responds. This controlled setup enables engineering teams to verify that their failover scripts and fallback paths function correctly under various stressful scenarios. Running these exercises on the simulator strengthens the overall reliability of automation pipelines, demonstrates compliance with relevant standards, and confirms that cloud-architecture plans hold up when individual components fail.

7.4 Training and Education Labs

The simulator is equally valuable for learning. Mastering cloud-orchestration platforms such as VCD demands practical, hands-on exposure; however, live environments are frequently too large, costly, or tightly governed for new staff to interact with freely. Schools, external training providers, and internal onboarding programs therefore struggle to deliver meaningful lab work without straining operational infrastructure. The simulator lowers that barrier by offering identical tools in an isolated, resettable space (18, 19). The simulator provides a safe, low-risk environment that closely replicates the production API of VCD. Trainees can log in, create vApps, manage networks, and explore resource hierarchies just as they would in a live environment. Because the simulator tracks state and responds dynamically, the learning experience feels almost indistinguishable from working with the actual cloud console.

Instructors can run the simulator on a laptop or host it in the cloud, tailoring it to any lesson plan and resetting it with a single command between classes. That flexibility accommodates both self-guided study and structured workshops, allowing cloud engineers to practice automation at their own pace without worrying about affecting production resources. The simulator adds convenience, reliability, and rapid feedback to nearly any workflow. Whether inserted into a CI/CD pipeline, used as an independent developer tool, woven into disaster-recovery tests, or deployed in training labs, it faithfully reproduces VCD behavior and accelerates the adoption of cloud orchestration.

8. Evaluation and Performance Metrics

The effectiveness of the VCD API simulator is assessed along three interrelated axes: fidelity to VMware vCloud Director Behavior, performance under varying load conditions, and the subjective usability experienced by developers and testers during their daily tasks. By examining these dimensions together, the evaluation determines whether the tool is both technically sound and practically advantageous in real development and test circuits.

As the figure below illustrates, the evaluation framework connects performance data, behavioral accuracy, and end-user satisfaction into a unified model. This holistic perspective allows stakeholders to measure not just how the simulator functions, but how effectively it supports development goals and reduces operational risks.

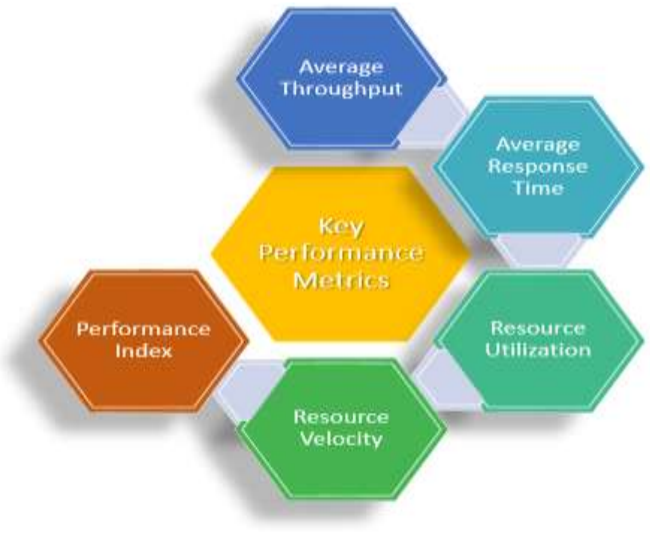


Figure 6: *Key Performance Metrics*

8.1 Fidelity to Real VCD Behaviour

A simulator gains value only when its outputs mirror those of the actual system it replicates (13). To test that criterion, the VCD simulator ran beside production VMware vCloud Director Instances, allowing every status code, payload identifier, and API pathway to be checked for correspondence. During these comparisons, identical commands—such as creating vApps, fetching catalog entries, establishing login sessions, or querying media resources—were issued to both the simulator and the live cluster, and the resulting JSON or XML documents were placed side by side for comparison. The comparison examined not only syntactic form but logical organization and semantic content against VMware's published API reference.

The outcomes confirmed that the simulator maintains a high degree of fidelity. Core elements, including hierarchical object relationships, pagination schemes for list endpoints, and metadata formatting, were reproduced with close correspondence to the reference system. Stateful actions such as vApp deployment and power-on followed established workflows, yielding expected state transitions and consistent timestamp generation throughout the test cycle. Deviations primarily arose in advanced situations, including granular network policy enforcement and the handling of long-running asynchronous jobs. These features were intentionally abstracted in the simulator to limit computational overhead and avoid unnecessary complexity. The documented differences were judged tolerable, since the simulator serves mainly as a test-and-automation facilitator, not as a drop-in substitute for production environments.

8.2 Performance Benchmarks

Performance experiments evaluated how the simulator scaled under representative loads, with particular focus on automated pipelines and scenarios that exercised high levels of concurrent testing. Principal metrics—endpoint latency, overall request throughput, and per-thread memory consumption—were recorded for each test class, enabling comparative analysis across low, medium, and peak workloads. Latency tests measured average response time for each API endpoint under idle and busy conditions. For lightweight calls, such as session login and metadata retrieval, the average response time remained below 50 milliseconds, even with dozens of concurrent users. For state-changing operations, such as vApp creation, which involve simulated object generation and state tracking, latency ranged from 70 to 120 milliseconds, depending on the payload size and response complexity.

Throughput was tested by subjecting the simulator to concurrent API calls via a load-testing framework. With fifty parallel users issuing a mixed pattern of read and write commands, the system sustained over fifteen hundred requests per second and produced only sporadic, non-influential errors. At peak load, response times rose slightly, yet the simulator remained responsive and stable for all measured flows. Memory footprint and CPU usage remained within acceptable thresholds, especially when deployed in Docker containers with specified resource limits. Redis-backed instances delivered stronger performance for short-lived sessions and frequent state lookups, while the MongoDB back end offered superior durability for longer-running, sequence-based simulations. Together, these benchmarks demonstrate that the simulator can comfortably support typical development, testing, and continuous integration workloads without becoming a bottleneck. The performance profile remains adequate for integration into automated pipelines and is scalable enough to accommodate multi-user environments.

The graph illustrates the behavior of the VMware vCloud Director (VCD) API simulator when subjected to steadily increasing numbers of virtual users, specifically, workloads consisting of 10, 25, and 50 concurrent transactions. Two primary metrics appear on the figure: latency and throughput. Latency is traced on the left Y-axis by red and orange lines and reflects average round-trip times for distinct classes of calls. For lightweight requests, such as logon and metadata read, latency remained remarkably stable, climbing only from 35 milliseconds to 48 milliseconds as the load doubled from ten to fifty users. In contrast, state-changing calls, including vApp provisioning, required more internal processing; here, round-trip times increased from 70 milliseconds to 120 milliseconds when the same user increment was applied.

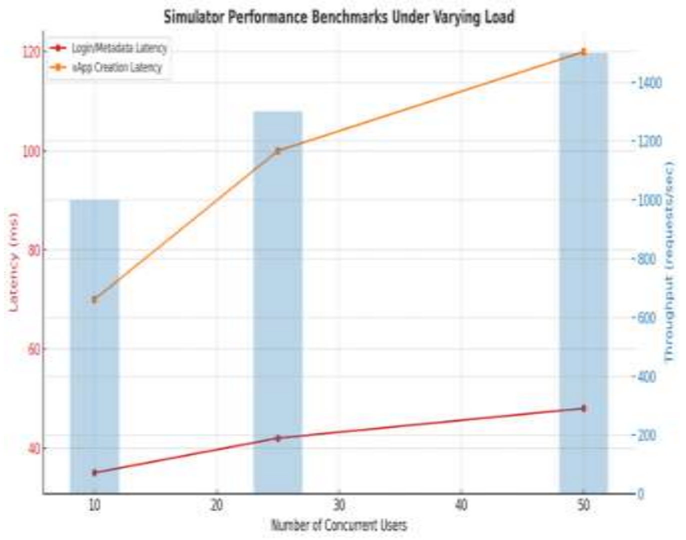


Figure 7: *Simulator Performance under Varying Load Conditions*

8.3 Developer Experience Feedback

The simulator's overall utility was also assessed through qualitative comments from developers, testers, and DevOps personnel, complementing the raw performance figures. Structured surveys were distributed after multiple integration cycles, especially to teams that had previously been hindered by restricted VCD visibility or by test beds that frequently lost stability. Respondents pointed to several immediate benefits. Foremost, practitioners valued the option to execute complete orchestration pipelines on local workstations, eliminating the need for VPN gateways or central cloud support. This single improvement accelerated debugging, allowing parallel runs and providing engineers with the room to experiment with early feature prototypes. Testers further remarked that the simulator made rare failure scenarios—such as token expiration, HTTP 500 responses, or intermittent timeouts—straightforward to generate and examine, something production-mimicking environments had long resisted.

The team also gathered quantitative evidence on automated testing coverage and defect capture (35). Across integration runs, every group logged a noticeable rise in executed test cases per build, a shift attributed to the simulator's easy availability and its broadly adjustable parameters. Throughout integration sprints, defect-catch ratios improved, resulting in fewer discrepancies advancing into staging or production. That trend was especially pronounced within error-handling flows and edge-case checks, areas that prior resource constraints had consistently left untested.

Several focus group sessions highlighted the value of comprehensive transaction logs, user-defined response templates, and built-in export options for OpenAPI specifications. These elements lowered the learning curve for novice testers and empowered seasoned engineers to fine-tune the environment for specialized scenarios at little extra cost. Taken together, the two strands of evaluation-technique accuracy and time-to-deliver in operational settings provide a clear verdict. The simulator reproducibly tracked core VCD patterns, sustained heavy loads without drift, and noticeably eased the developer-QA feedback loop. By reducing provisioning overhead, expanding the scope of automation, and accelerating cycle times, it now meets the daily demands of cloud orchestration testing. A key measure of the simulator’s success was its impact on developer productivity and testing quality. As detailed in Table 4, developers reported significant improvements in local testing, faster defect detection, and enhanced coverage for edge-case scenarios.

Table 4: *Developer Experience Feedback:*

Aspect	**Feedback / Observations**
Local Execution	Enabled full orchestration pipelines on local machines; eliminated VPN/cloud dependency.
Debugging & Prototyping	Accelerated debugging; allowed parallel runs and early feature experimentation.
Failure Scenario Simulation	Simplified testing of rare cases (e.g., token expiration, HTTP 500, timeouts).

Aspect	Feedback / Observations
Automated Testing Coverage	Increased executed test cases per build due to availability and configurability.
Defect Detection	Higher defect-catch ratios, especially in error-handling and edge-case logic.
Support Tools	Provided detailed logs, custom response templates, and OpenAPI export options.
Learning Curve	Lowered for new testers; enhanced scenario flexibility for experienced users.
Operational Value	Improved evaluation accuracy, reduced delivery time, and optimized dev-QA feedback loop.
System Reliability	Maintained stability under load and reliably emulated core VCD API patterns.

9. Limitations and Future Enhancements

9.1 Known Limitations

Although the VCD API simulator is reliable for training and testing, it has several limitations stemming from deliberate architectural choices and design compromises made to keep the implementation straightforward. These weaknesses do not typically disrupt standard testing, but they can complicate more complex scenarios, so users should consider them before applying the tool in demanding settings. Concurrency and session handling represent one of the most noticeable constraints. The simulator does remember individual sessions and preserves state across authenticated request cycles, but it has not been fine-tuned for environments where large numbers of users execute overlapping calls simultaneously. In contrast to a fully featured API gateway, it omits per-tenant memory partitions, token revocation lists, and built-in load balancing, all of which strengthen isolation and scaling. As a result, running the simulator under heavy parallel loads that alter shared data can still introduce conflicts or race conditions unless the test sequence is carefully organized.

Its support for the full VCD resource lifecycle remains incomplete (10). Although the simulator manages the standard phases for virtual applications—creation, deployment, power-on, and deletion—it overlooks operational controls such as resource locking, job queues, and time-sensitive state transitions. In an actual VCD deployment, tasks can be queued, blocked, or aborted depending on role permissions, current states, and backend dependencies. The simulator, however, adopts an optimistic model: as long as the mock state logically permits an action, it executes immediately. Because of this design choice, edge cases—such as attempting to delete a vApp while another command is pending—are not accurately reproduced.

Error handling is similarly basic. The fault-injection tool can emulate timeouts, service-level errors, and malformed messages, yet it stops short of modeling nuanced or cascading failures that arise in large, distributed systems. Although token-based authentication is effective, the simulator only partially implements multi-role access control and the whole hierarchy of permissions. None of these omissions invalidate basic test cycles, but they do impose limits when users try to mimic sophisticated orchestration timing, enforce detailed policies, or reproduce production-like, multithreaded workloads.

9.2 Future Work

The development team has identified multiple enhancement opportunities aimed at overcoming existing limitations and broadening the simulator's utility across the entire software development lifecycle. First on the agenda is a web-based graphical user interface (GUI). A well-designed dashboard would allow users to visualize API endpoints, examine real-time state data (such as active sessions or allocated virtual applications), adjust response templates, and simulate fault conditions through point-and-click actions. By reducing dependence on command-line syntax, the GUI would lower the entrance barrier for quality-assurance testers, educators, and DevOps engineers who routinely prefer visual instrumentation during verification.

The second major expansion targets tighter integration with popular Infrastructure-as-Code (IaC) frameworks, most notably Terraform and Ansible. Because contemporary teams rely on these tools to standardize provisioning pipelines across heterogeneous cloud providers, creating dedicated provider modules or plugins linked to the simulator would enable developers to validate their declarative configurations before committing resources externally. Such a testing stage would expose syntactic errors, reveal hidden assumptions, and furnish higher confidence in template correctness, ultimately streamlining pre-deployment governance and reducing unexpected charges from live environments. Plans are being finalized to strengthen the internal simulation engine by adding several new error-handling features. Among these are linked failure events, randomized fault injection, models of resource exhaustion, and emulation of degraded performance across subsystems. Together, these capabilities will enable developers to subject a service to realistic, cascading faults and test resilience patterns, such as retries, graceful degradation, and fallback procedures. The resulting tests will provide deeper insight into how production code behaves under adverse yet plausible conditions. The workflow engine will also gain a new layer of configurability, permitting users to chain operations, introduce conditional waits, and express timing rules in a domain-specific syntax. A typical use case might specify that an API call for vApp creation remains in the "deploying" state for exactly thirty seconds before moving to "ready," thereby simulating genuine provisioning latencies observed in cloud environments. By allowing administrators to sequence calls and associate delays or blockers with distinct failure states, the extended engine supports the rigorous staging of complex orchestration scenarios and the precise validation of time-sensitive automation logic. The current release of the VCD API simulator already serves a wide range of development and evaluation tasks, yet it remains in active evolution (4). Planned updates will refine the user interface, deepen integration with standard toolchains, and enhance error emulation, making the simulator a more rounded offering. These additions aim to bridge the gap between straightforward mock tests and the complex behavior observed in production cloud environments, thereby encouraging wider adoption and greater trust in automated infrastructure deployments.

As the figure below illustrates, this roadmap strategically targets key layers of the development process, aiming to position the simulator as a foundational tool in automated infrastructure testing.

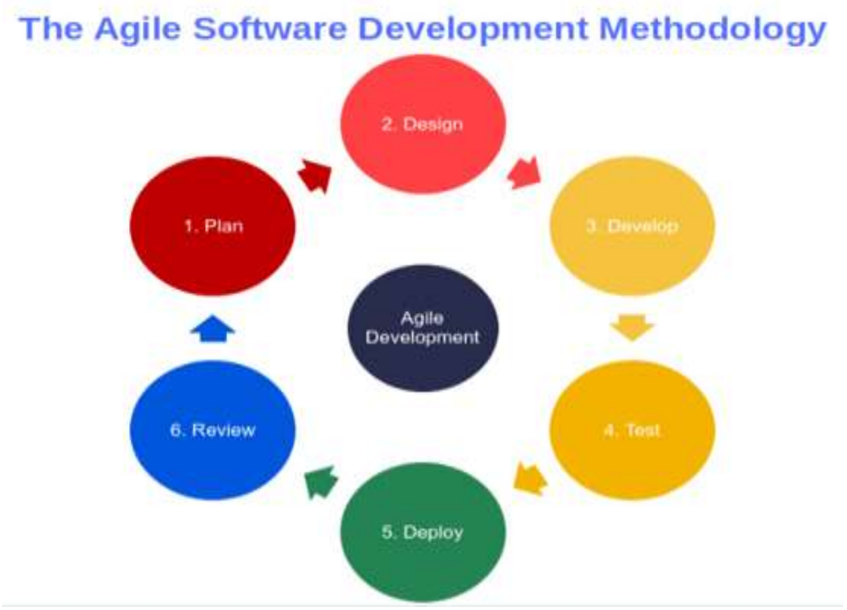


Figure 8: *software-development-strategy*

10. Recommendations

The ongoing work on the VCD API simulator shows clear promise for strengthening test automation within cloud engineering. To amplify that benefit across sectors and settings, the following focused recommendations are offered to practitioners, organizations, educators, and software developers.

10.1 For DevOps and QA Teams

Teams that rely on VMware vCloud Director should add the simulator to their DevOps pipelines as soon as practical (9). Swapping or supplementing live tests with a simulated environment accelerates feedback loops, reduces risk, and expands the reach of automation. It is advisable to run the simulator during nightly builds, regression suites, and integration checks, particularly where orchestration code calls VCD APIs. Quality assurance engineers should consistently utilize the simulator's fault-injection features when performing chaos tests. By simulating API timeouts, unauthorized calls, and sluggish responses, testers can strengthen the underlying automation scripts and optimize the backing infrastructure for uncommon but possible operational anomalies.

10.2 For cloud architects and infrastructure engineers

Cloud architects may therefore treat the simulator as an iterative mockup environment in which orchestration drafts and multi-tenant blueprints can be field-tested. Executing templates against the simulator before promotion to production VCD confirms provisioning logic, verifies API ordering, and checks policy compliance without affecting live customers. Infrastructure engineers, in turn, should advocate for the broader adoption of the tool across their infrastructure-as-code pipelines. By submitting Terraform or Ansible modules that mirror the simulator's input schema, teams narrow the gap between code commits and runtime observations, significantly reducing misconfiguration errors when the software reaches production.

10.3 For Enterprises and Managed Service Providers

Enterprises running multiple VCD clusters or supplying vCloud services to customers should consider using the simulator during internal testing and onboarding activities. By trying integrations against the simulator before production access is granted, teams validate API calls, shorten ramp-up times, and ease the workload on live systems. Managed service providers may also adopt the simulator in their service-assurance workflows, allowing vendors and partners to verify automation scripts and configurations before they enter sensitive environments.

10.4 For Educators and Training Institutions

Educators and cloud trainers can weave the simulator into DevOps, cloud computing, and virtualization programs. Its quick setup and behavior, closely aligned with VMware vCloud Director, make it well-suited for lab sessions, workshops, and self-guided study. Learners practice genuine cloud-orchestration tasks without needing expensive hardware or endangering running services. Instructors are encouraged to package the simulator with ready-made exercises, step-by-step guides, and OpenAPI documentation, thereby creating a safe sandbox where students can explore and deepen their understanding of the material.

10.5 For Simulator Developers and Maintainers

To ensure the simulator remains timely and helpful, the development team should advance the roadmap items listed in Section 9.2. Key priorities include a web-based graphical user interface to simplify configuration, built-in support for Terraform and Ansible, and richer error-emulation modules (1). Adding multi-session concurrency, nested API workflow support, and refined token-driven role management will broaden the tool's relevance to enterprise-scale deployments. Publishing the simulator as an open-source asset deserves serious consideration. An open model invites community patches, accelerates peer review, and promotes widespread adoption. Hosting practical example projects alongside user-contributed case studies will inform future enhancements while illustrating the simulator's value to prospective teams.

As the figure below illustrates, fostering an open, extensible, and community-supported simulator aligns with modern software development best practices and enhances long-term sustainability.

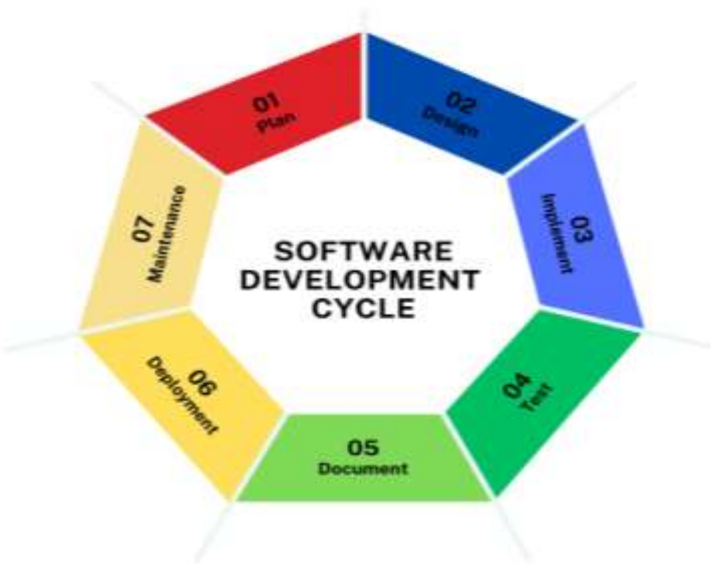


Figure 9: *software-development-process*

11. Conclusion

The creation of a dedicated simulator for VMware vCloud Director API calls represents a significant advancement in cloud orchestration testing. Tailored for developers, testers, and DevOps operators, the tool provides a safe, efficient, and inexpensive substitute for live-system validation. By accurately modeling authentication, resource provisioning, and lifecycle functions, it enables thorough workflow verification without the hazards or access limitations typically found in production environments. The simulator surpasses standard mocking frameworks by incorporating capabilities specifically designed for cloud orchestration scenarios. Among these are stateful tracking of virtual resources, multi-step API choreography, and on-the-fly payload generation that mirrors parameters found in production requests. Its built-in fault-injection functions enable developers to evaluate error-handling paths and measure resilience, positioning the simulator as a critical asset for constructing and verifying automation pipelines. Because it reproduces live conditions in a contained manner, the tool enhances both safety and throughput throughout the cloud development lifecycle.

Its effect on cloud-development practices is already noticeable. By removing the dependence on live VDC infrastructure, the simulator reduces the entry cost of infrastructure-as-code, continuous delivery, and automated testing pipelines. DevOps teams in particular gain from being able to run API workflows, explore edge cases, and test automation logic in completely isolated targets. Because the simulator is packaged as a lightweight container, it can be dropped straight into CI/CD chains on systems like GitHub Actions or Jenkins, ensuring every build receives fresh validation before code leaves the branch. The early detection of defects not only speeds up the pipeline but also steadies production by revealing inconsistencies before they occur.

The simulator also preserves key orchestration mechanics—session management, endpoint emulation, and lifecycle simulation—that modern iterative teams require. In complex multi-tenant clouds, where live tests may be impractical or unsafe, the tool provides repeatable and predictable behavior across a wide range of operating conditions. Looking ahead, this simulator is likely to find roles well beyond the present experiments it supports. Within large companies, it could be featured in internal validation suites that guide the secure onboarding of new automation teams and external partners. Similarly, cloud providers and managed-service firms may use the tool to verify third-party integrations before granting production access, thereby enhancing reliability and reducing ongoing support demands. For students and instructors, the performer serves as an affordable alternative to full-scale cloud laboratories, allowing learners to practice orchestration tasks with real commands without incurring the expense or operational burden of live hardware.

A series of planned updates—among them a polished web interface, tighter integration with Terraform and Ansible, and more robust error modeling—are expected to expand the simulator's capacity further. When finished, these features should make the tool easier to navigate, more responsive to infrastructure-as-code workflows, and better able to recreate intricate runtime situations involving interdependencies, latencies, and conditional branches. The VCD API simulator fills a crucial gap in cloud engineering by providing programmers with a safe, repeatable, and realistic environment for validating orchestration logic. By using the simulator, DevOps teams can iterate more quickly, reduce testing costs, and run bold experiments without jeopardizing production stability. Given the accelerating shift toward scalable, cloud-native architectures, instruments of this kind will prove vital for maintaining infrastructure agility and resilience. Ongoing improvements and broader adoption of the tool are expected to shape patterns in how future cloud services are designed, tested, and rolled out across the sector.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

References

[1] Aranda, L. A., Ruano, O., Garcia-Herrero, F., & Maestro, J. A. (2021). Reliability Analysis of ASIC Designs With Xilinx SRAM-Based FPGAs. IEEE Access, 9, 140676-140685. https://doi.org/10.1109/ACCESS.2021.3119633

[2] Babashamsi, P., Yusoff, N. I. M., Ceylan, H., Nor, N. G. M., & Jenatabadi, H. S. (2016). Evaluation of pavement life cycle cost analysis: Review and analysis. International Journal of Pavement Research and Technology, 9(4), 241-254. https://doi.org/10.1016/j.ijprt.2016.08.004

[3] Baur, D., Seybold, D., Griesinger, F., Tsitsipas, A., Hauser, C. B., & Domaschka, J. (2015, December). Cloud orchestration features: Are tools fit for purpose?. In 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC) (pp. 95-101). IEEE. https://doi.org/10.1109/UCC.2015.25

[4] Bennett, B. E. (2021, April). A practical method for API testing in the context of continuous delivery and behavior driven development. In 2021 IEEE international conference on software testing, verification and validation workshops (ICSTW) (pp. 44-47). IEEE. https://doi.org/10.1109/ICSTW52544.2021.00020

[5] Bialek, J., Ciapessoni, E., Cirio, D., Cotilla-Sanchez, E., Dent, C., Dobson, I., ... & Wu, D. (2016). Benchmarking and validation of cascading failure analysis tools. IEEE Transactions on Power Systems, 31(6), 4887-4900. https://doi.org/10.1109/TPWRS.2016.2518660

[6] Casas, S., Cruz, D., Vidal, G., & Constanzo, M. (2021, November). Uses and applications of the OpenAPI/Swagger specification: a systematic mapping of the literature. In 2021 40th International Conference of the Chilean Computer Science Society (SCCC) (pp. 1-8). IEEE. https://doi.org/10.1109/SCCC54552.2021.9650408

[7] Chavan, A. (2022). Importance of identifying and establishing context boundaries while migrating from monolith to microservices. Journal of Engineering and Applied Sciences Technology, 4, E168. http://doi.org/10.47363/JEAST/2022(4)E168

[8] Chavan, A. (2024). Fault-tolerant event-driven systems: Techniques and best practices. Journal of Engineering and Applied Sciences Technology, 6, E167. http://doi.org/10.47363/JEAST/2024(6)E167

[9] Dakic, V., Chirammal, H. D., Mukhedkar, P., & Vettathu, A. (2020). Mastering KVM virtualization: design expert data center virtualization solutions with the power of Linux KVM. Packt Publishing Ltd.

[10] Del Savio, A. A., Vidal Quincot, J. F., Bazán Montalto, A. D., Rischmoller Delgado, L. A., & Fischer, M. (2022). Virtual Design and Construction (VDC) Framework: A Current Review, Update and Discussion. Applied sciences, 12(23), 12178. https://doi.org/10.3390/app122312178

[11] Dhanagari, M. R. (2024). MongoDB and data consistency: Bridging the gap between performance and reliability. Journal of Computer Science and Technology Studies, 6(2), 183-198. https://doi.org/10.32996/jcsts.2024.6.2.21

[12] Dhanagari, M. R. (2024). Scaling with MongoDB: Solutions for handling big data in real-time. Journal of Computer Science and Technology Studies, 6(5), 246-264. https://doi.org/10.32996/jcsts.2024.6.5.20

[13] Eckhart, M., & Ekelhart, A. (2018, January). A specification-based state replication approach for digital twins. In Proceedings of the 2018 workshop on cyber-physical systems security and privacy (pp. 36-47). https://doi.org/10.1145/3264888.3264892

[14] Ehsan, A., Abuhaliqa, M. A. M., Catal, C., & Mishra, D. (2022). RESTful API testing methodologies: Rationale, challenges, and solution directions. Applied Sciences, 12(9), 4369. https://doi.org/10.3390/app12094369

[15] Franz, T., Seidl, C., Fischer, P. M., & Gerndt, A. (2022). Utilizing multi-level concepts for multi-phase modeling: Context-awareness and process-based constraints to enable model evolution. Software and Systems Modeling, 21(4), 1665-1683. https://link.springer.com/article/10.1007/s10270-021-00963-1

[16] Goel, G., & Bhramhabhatt, R. (2024). Dual sourcing strategies. International Journal of Science and Research Archive, 13(2), 2155. https://doi.org/10.30574/ijsra.2024.13.2.2155

[17] Jarecki, S., Jubur, M., Krawczyk, H., Shirvanian, M., & Saxena, N. (2018). Two-Factor Password-Authenticated Key Exchange with End-to-End Password Security. Cryptology ePrint Archive. https://ia.cr/2018/033

[18] Karwa, K. (2023). AI-powered career coaching: Evaluating feedback tools for design students. Indian Journal of Economics & Business. https://www.ashwinanokha.com/ijeb-v22-4-2023.php

[19] Karwa, K. (2024). Navigating the job market: Tailored career advice for design students. International Journal of Emerging Business, 23(2). https://www.ashwinanokha.com/ijeb-v23-2-2024.php

[20] Konneru, N. M. K. (2021). Integrating security into CI/CD pipelines: A DevSecOps approach with SAST, DAST, and SCA tools. International Journal of Science and Research Archive. Retrieved from https://ijsra.net/content/role-notification-scheduling-improving-patient

[21] Kumar, A. (2019). The convergence of predictive analytics in driving business intelligence and enhancing DevOps efficiency. International Journal of Computational Engineering and Management, 6(6), 118-142. Retrieved from https://ijcem.in/wp-content/uploads/THE-CONVERGENCE-OF-PREDICTIVE-ANALYTICS-IN-DRIVING-BUSINESS-INTELLIGENCE-AND-ENHANCING-DEVOPS-EFFICIENCY.pdf

[22] Kumar, P. S., Emfinger, W., Karsai, G., Watkins, D., Gasser, B., & Anilkumar, A. (2016). ROSMOD: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based

software using ROS. Electronics, 5(3), 53. https://doi.org/10.3390/electronics5030053

[23] Morchid, A., Alblushi, I. G. M., Khalid, H. M., El Alami, R., Sitaramanan, S. R., & Muyeen, S. M. (2024). High-technology agriculture system to enhance food security: A concept of smart irrigation system using Internet of Things and cloud computing. Journal of the Saudi Society of Agricultural Sciences. https://doi.org/10.1016/j.jssas.2024.02.001

[24] Nieto, M., Senderos, O., & Otaegui, O. (2021). Boosting AI applications: Labeling format for complex datasets. SoftwareX, 13, 100653. https://doi.org/10.1016/j.softx.2020.100653

[25] Nyati, S. (2018). Revolutionizing LTL carrier operations: A comprehensive analysis of an algorithm-driven pickup and delivery dispatching solution. International Journal of Science and Research (IJSR), 7(2), 1659-1666. Retrieved from https://www.ijsr.net/getabstract.php?paperid=SR24203183637

[26] Raju, R. K. (2017). Dynamic memory inference network for natural language inference. International Journal of Science and Research (IJSR), 6(2). https://www.ijsr.net/archive/v6i2/SR24926091431.pdf

[27] Ronen, E., Gillham, R., Genkin, D., Shamir, A., Wong, D., & Yarom, Y. (2019, May). The 9 lives of Bleichenbacher's CAT: New cache attacks on TLS implementations. In 2019 IEEE Symposium on Security and Privacy (SP) (pp. 435-452). IEEE. https://doi.org/10.1109/SP.2019.00062

[28] Sardana, J. (2022). The role of notification scheduling in improving patient outcomes. International Journal of Science and Research Archive. Retrieved from https://ijsra.net/content/role-notification-scheduling-improving-patient

[29] Singh, V. (2022). Visual question answering using transformer architectures: Applying transformer models to improve performance in VQA tasks. Journal of Artificial Intelligence and Cognitive Computing, 1(E228). https://doi.org/10.47363/JAICC/2022(1)E228

[30] Singh, V. (2023). Enhancing object detection with self-supervised learning: Improving object detection algorithms using unlabeled data through self-supervised techniques. International Journal of Advanced Engineering and Technology. https://romanpub.com/resources/Vol%205%20%2C%20No%201%20-%2023.pdf

[31] Sukhadiya, J., Pandya, H., & Singh, V. (2018). Comparison of Image Captioning Methods. INTERNATIONAL JOURNAL OF ENGINEERING DEVELOPMENT AND RESEARCH, 6(4), 43-48. https://rjwave.org/ijedr/papers/IJEDR1804011.pdf

[32] Svensson, A. (2024). What is the best API from adeveloper’s perspective?: Investigation of API development with fintechdevelopers in the spotlight. https://www.diva-portal.org/smash/get/diva2:1865779/FULLTEXT02

[33] Tiwari, D., Monperrus, M., & Baudry, B. (2024). Mimicking production behavior with generated mocks. IEEE Transactions on Software Engineering. https://doi.org/10.1109/TSE.2024.3458448

[34] Ugwueze, V. U., & Chukwunweike, J. N. (2024). Continuous integration and deployment strategies for streamlined DevOps in software engineering and application delivery. Int J Comput Appl Technol Res, 14(1), 1-24. http://www.ijcat.com/

[35] Wang, Y., Mäntylä, M. V., Liu, Z., & Markkula, J. (2022). Test automation maturity improves product quality—Quantitative study of open source projects using continuous integration. Journal of Systems and Software, 188, 111259. https://doi.org/10.1016/j.jss.2022.111259