

Unfulfilled Promises: LLM-Based Detection of OS Compatibility Issues in Infrastructure as Code

GEORGIOS-PETROS DROSOS, ETH Zurich, Switzerland

GEORGIOS ALEXOPOULOS, University of Athens, and National Infrastructures for Research and Technology, Greece

THODORIS SOTIROPOULOS, ETH Zurich, Switzerland

DIMITRIS MITROPOULOS, University of Athens, and National Infrastructures for Research and Technology, Greece

ZHENDONG SU, ETH Zurich, Switzerland

Modern infrastructures rely on Infrastructure as Code (IaC) systems to keep complex deployments consistent, reproducible, and scalable at production scale. The reliability of these infrastructures, however, depends on the correctness of their building blocks, which are reusable components (modules) that each performs a dedicated task, such as installing a package, managing an OS user, or configuring a service, and reconciling its state with the desired specification. A central promise of these components is *portability*: a specification written once should correctly manage the targeted resource on every OS the IaC component supports. When this property is violated, defects can propagate across entire infrastructures, causing outages, security vulnerabilities, and costly misconfigurations.

In this work, we introduce **cROSSIBLE**, the first automated framework for cross-OS testing of IaC modules. **cROSSIBLE** leverages large language models (LLMs) to synthesize and repair integration tests from structured module documentation, and executes them across 13 versions of 8 major Linux distributions. While our techniques are generally applicable to different IaC systems, we instantiate and evaluate them on Ansible, the most widely used IaC framework for managing individual servers. Evaluation across 259 popular Ansible modules demonstrates both effectiveness and real-world impact. In just 12 hours of testing, **cROSSIBLE** uncovered 36 previously unknown bugs, including 22 portability violations. In total, 27 issues have been confirmed by maintainers, with 11 already fixed. The discovered issues range from crashes to dangerous soundness defects where modules reported success despite leaving systems misconfigured. Beyond bug discovery, **cROSSIBLE** improved the code coverage of Ansible modules by 12.3% on average, systematically exercising OS-specific code paths that existing tests missed.

CCS Concepts: • Software and its engineering → Software testing and debugging.

Additional Key Words and Phrases: infrastructure as code, state reconciliation, cross-platform testing, Ansible

1 Introduction

Modern software systems run on large, heterogeneous infrastructures that must be provisioned and maintained consistently across environments. The *Infrastructure as Code (IaC)* paradigm addresses this challenge by shifting operational tasks into declarative specifications, thereby reducing the risk of errors and enabling reproducible deployments. In production, IaC reliability is critical: defects in infrastructure logic have caused outages, security incidents, and costly downtime [1, 27, 37, 60].

In this work, we focus on Ansible [40], one of the most widely adopted IaC systems in industry [29]. Ansible programs, called *playbooks*, are written in a declarative domain-specific language (YAML) and specify the desired state of a computing infrastructure. Playbooks are composed of invocations of Ansible *modules*, which encapsulate the logic for configuring individual system

Authors' Contact Information: [Georgios-Petros Drosos](#), ETH Zurich, Zurich, Switzerland, gdrosos@student.ethz.ch; [Georgios Alexopoulos](#), University of Athens, and National Infrastructures for Research and Technology, Athens, Greece, grgalex@ba.uoa.gr; [Thodoris Sotiropoulos](#), ETH Zurich, Zurich, Switzerland, theodoros.sotiropoulos@inf.ethz.ch; [Dimitris Mitropoulos](#), University of Athens, and National Infrastructures for Research and Technology, Athens, Greece, dimitro@ba.uoa.gr; [Zhendong Su](#), ETH Zurich, Zurich, Switzerland, zhendong.su@inf.ethz.ch.

resources. Modules are to playbooks what shared libraries are to compiled programs: a single defect in an Ansible module can cascade into thousands of failing playbook deployments [48].

Each module exposes an API that lets users describe the intended system state through key-value parameters. At runtime, the module inspects the current system state (e.g., running services, configuration files) and applies the necessary changes to reconcile it with the desired state, in a manner analogous to the operator pattern in Kubernetes [35].

A central promise of modules is *portability* across their supported operating systems (OSes) [23], achieved by abstracting away OS-specific differences from users and ensuring consistent behavior through OS-specific logic in their implementation. *But do Ansible modules truly fulfill their promise?* A recent study [23] shows that over 20% of IaC bugs are *OS compatibility issues*, failures that manifest *only* on a subset of supported OSes, or only on specific OS versions. Motivated by these findings, this paper seeks to provide answers to this question and detect violations of the portability property.

An example OS compatibility bug: Figure 1 illustrates an OS compatibility bug (detected in the context of our work) in the `community.general.cronvar` module, which manages environment variables in user crontabs. The scenario involves a system administrator writing a playbook to ensure that the crontab includes an environment variable `foo` with an empty value. The module implements this by writing the variable to the cron daemon. However, different OSes use different cron implementations: Fedora’s `cronie` [14] accepts the entry `foo=` as valid, while Debian’s `vixie-cron` [46] rejects it, eventually causing the playbook to terminate unexpectedly. The fix was to modify the module to write `foo=""` instead of `foo=` in the crontab, ensuring compatibility with both OSes.

Despite being prevalent [23], no approach exists for detecting OS compatibility bugs in Ansible modules. In fact, the vast majority of research has focused on playbook (program) testing [15, 30, 43, 44, 47–54, 56], with only Hassan et al. [32] focusing on modules, but from an OS agnostic scope.

Approach: Integration tests [8] are the primary mechanism used to validate the correctness of Ansible modules. But do they assess modules’ portability? To identify existing gaps and assess the effectiveness of the upstream integration tests in exercising OS-specific behavior, we perform a study of 434 widely used Ansible modules. The findings of our study then guide the design of `CROSSIBLE`, the first automated framework for detecting OS compatibility bugs in Ansible modules.

Detecting such bugs requires more than simply rerunning existing tests across multiple OSes. We first introduce a formal model of OS compatibility bugs, which defines when executions across OSes are comparable and provides a foundation for reasoning about portability. From this model, three challenges emerge: (a) exploring diverse initial states that satisfy module dependencies, (b) generating meaningful module invocations that respect each module’s API, and (c) validating final states against the given module invocations.

To address these challenges, `CROSSIBLE` leverages large language models (LLMs) to synthesize new integration tests directly from module documentation, enriched by insights from the empirical study. `CROSSIBLE` uses a two-stage prompting process: the first stage generates natural-language test scenarios aimed at uncovering OS compatibility bugs, while the second translates them into

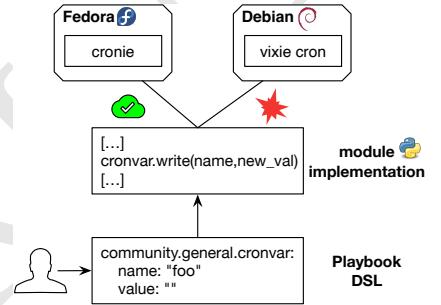


Fig. 1. Bug found by our approach in the `cronvar` [22] module of the `community.general` collection. The same playbook succeeds on Fedora, but fails on Debian.

executable playbooks. The generated playbooks are validated, automatically repaired if needed, and executed across multiple Linux distributions. Execution outcomes are then differentially compared, with divergences filtered to exclude errors from improper setup tasks, leaving only genuine bugs.

Key results: Our empirical study shows that existing module integration tests (1) insufficiently exercise all supported OSes, mainly due to maintenance and continuous integration (CI) costs; (2) traverse nearly identical code paths across OSes (mean code coverage 80%, maximum variation 0.73%); (3) rarely test modules on invalid inputs (these account for only 12.5% of test cases per module); and (4) rely mainly on return values as oracles rather than also checking the resulting system state (state-based assertions appear in only 10.8% of test cases per module).

Guided by these findings, we design and implement cROSsible, which we evaluate on 259 popular Ansible modules. In just 12 hours, cROSsible uncovers 36 previously unknown bugs (22 OS compatibility, 14 OS-agnostic), of which 27 are confirmed and 11 already fixed. Additionally, it improves the average code coverage across modules by 12.3 points, from 63.2% to 75.5%.

Contributions: This paper makes the following contributions:

- A formal model for reasoning about OS compatibility bugs in IaC systems.
- An empirical study revealing weaknesses in existing Ansible module integration test suites.
- cROSsible, a practical and scalable LLM-based framework that synthesizes integration tests from module documentation to uncover OS compatibility bugs.
- An extensive evaluation of cROSsible, covering several aspects, including bug-finding capability, code coverage, and bug characteristics. Overall, cROSsible uncovered 36 bugs in popular Ansible modules, of which 27 are confirmed and 11 have been fixed. Notably, 13 bugs have been found in modules included in the main Ansible repository and maintained by the core Ansible developers.

2 Background and Problem Definition

IaC and Ansible: *Infrastructure as Code* (*IaC*) manages and provisions computing infrastructure (e.g., servers, networks) through high-level configuration files rather than manual operations. Users write declarative IaC programs in domain-specific languages (DSLs) that specify the desired system state. The IaC system then parses and executes these programs, invoking *building blocks* that reconcile system resources (e.g., packages, services, files) with the declared state.

This work focuses on Ansible, the most widely used IaC system for server management. In Ansible, IaC programs are called *playbooks* written in YAML format, and their building blocks are *modules*, which are small Python programs that perform specific tasks such as installing software, configuring services, or managing files. Each module invocation inside a playbook is called a *task*. A task can result in either **Success** or **Failure**. Success outcomes include: `ok` (the resource is already in the desired state), `changed` (the resource was modified and reconciled to the desired state), and `skipped` (the module was not applicable). Failure outcomes incorporate: `failed` (the task could not be completed because of an unexpected error) and `unreachable` (the target host is not reachable) values. When a task fails, Ansible halts playbook execution immediately, unless errors are explicitly suppressed via the `ignore_errors` flag.

Figure 2 presents an Ansible playbook that installs and sets up an Nginx server. The playbook includes three tasks. First, `ansible.builtin.package` installs the `nginx` package on the target

```

1 - name: "Install nginx"
2   ansible.builtin.package:
3     name: "nginx"
4     state: present
5
6 - name: "Deploy nginx configuration"
7   ansible.builtin.copy:
8     src: "files/nginx.conf"
9     dest: "/etc/nginx/nginx.conf"
10
11 - name: "Ensure nginx is running"
12   ansible.builtin.service:
13     service_name: "nginx"
14     state: started
15     enabled: yes

```

Fig. 2. Configure nginx using Ansible.

```

1  # 1. Global Setup: Install Node.js system package
2  - name: "Install Node.js on Debian" # Case A
3  ansible.builtin.apt:
4      name: nodejs
5      state: present
6  when: ansible_facts['os_family'] == "Debian"
7
8  - name: "Install Node.js on Arch" # Case B
9  ansible.builtin.pacman:
10     name: nodejs
11     state: present
12 when: ansible_facts['os_family'] == "Archlinux"
13
14 # 2. Test Case Setup: Clear npm packages
15 - name: "Remove existing node_modules"
16  ansible.builtin.file:
17      path: "/usr/lib/node_modules"
18      state: absent
19
20  # 3. Module Invocation
21  - name: "Install 'sqlite3' package"
22    community.general.npm:
23        state: present
24        name: "sqlite3"
25        register: npm_result
26
27  # 4. Validation: Check module's return value
28  - name: "Assert package installation succeeded"
29    ansible.builtin.assert:
30        that:
31            - npm_result is success

```

Fig. 3. Test for `community.general.npm` [12], showing all test phases, conditional invocation for cross-OS setup, and validation of the module’s reported result.

system (lines 1–4). Next, `ansible.builtin.copy` deploys the Nginx configuration file to the desired destination (lines 6–9). Finally, `ansible.builtin.service` ensures that the Nginx service is both started and enabled on boot (lines 11–15). Each module may receive arguments that influence its functionality. For example, `ansible.builtin.package` takes two arguments (lines 3 and 4): `name: "nginx"`, which specifies the package to manage, and `state: present`, which specifies that the package should exist on the target system.

Ansible modules and playbooks are distributed through a specific packaging format called *collections*. Collections are published on *Galaxy*, Ansible’s official package registry [3]. Ansible also offers some collections built into its runtime, such as `ansible.builtin`, similar to what Python does with modules such as `os` and `math`.

Testing in Ansible: Ansible collections primarily employ three types of testing: *sanity tests*, *unit tests*, and *integration tests* [10]. Sanity tests perform lightweight static code analysis to enforce coding standards and detect common issues (e.g., ill-formed YAML files), while unit tests verify the behavior of individual parts of the modules’ implementation. In contrast, integration tests validate functional correctness by executing modules in *realistic* environments and verifying both their status and effects on system state.

The integration test suite of a module is typically run as a playbook that includes a series of tasks exercising the module under test in various scenarios. Figure 3 shows an integration test adapted from the `community.general.npm` [12] module’s test suite. This test case validates that the module successfully installs a specified `npm` package on the target system. An integration test case in Ansible follows the workflow below:

1. *Global setup.* These are tasks that install dependencies and set up the necessary environment to invoke the module under test. For example, the `community.general.npm` module requires Node.js. Therefore, before invoking the module, Node.js must be installed on the target system (Figure 3, lines 1–12). Setup tasks may include conditional statements to account for different OS distributions (e.g., Figure 3, lines 6 and 9). Notably, global setup tasks are often *shared* between different test cases.

2. *Test case setup.* These are tasks that establish the system state required for a specific invocation of the module under test. For example, a scenario that verifies whether `community.general.npm` installs `sqlite3` must begin with a state where no `npm` package is installed on the target system (Figure 3, lines 14–18). These tasks are scoped to individual test cases and ensure that the module operates on a clean and controlled initial state.

3. *Module invocation.* These tasks execute the module under test with various parameters to simulate different usage scenarios. For example, in Figure 3, the `community.general.npm` module

```

1  # 2a. Test Case Setup
2  - name: "Create user with home /tmp/oldhome"
3  ansible.builtin.user:
4    name: movefuser
5    home: /tmp/oldhome
6    create_home: yes
7
8  # 2b. Test Case Setup
9  - name: "Create a file in the original home"
10 ansible.builtin.copy:
11   dest: /tmp/oldhome/testfile.txt
12   content: "old home"
13
14
15
16
17
18
19  # 3. Module Invocation
20  - name: "Move home directory"
21  ansible.builtin.user:
22    name: movefuser
23    home: /tmp/newhome
24    move_home: true
25
26  # 3a. Validation: Probe state
27  - name: "Stat file in new home"
28  ansible.builtin.stat:
29    path: /tmp/newhome/testfile.txt
30    register: movedfile
31
32  # 3b. Validation: Assert on probed state
33  - name: "Ensure home moved"
34  ansible.builtin.assert:
35    that:
36      - movedfile.stat.exists

```

Fig. 4. Test generated by our tool for the built-in user [57] module, where explicit state validation uncovers an OS compatibility bug on Alpine Linux.

is invoked with two named arguments specifying that the `npm` package `sqlite3` should be installed on the system (lines 20–24).

4. *Validation*. Upon invoking the module under test, an integration test ensures that the module behaves as expected. This is done by either: (a) checking the task’s `status` (e.g., **Success** or **Failure**; Figure 3, lines 27–30) using specific testing utilities, such as the `assert` module, or (b) running additional tasks to validate the resulting system state directly (more details in Figure 4, lines 21–31). If an integration test fails with an `Assertion Error`, it failed during validation; otherwise it failed in a previous phase.

2.1 OS Compatibility Bugs in Ansible Modules

A key promise of IaC systems such as Ansible is *portability* [23]: Ansible provides many modules that are intended to operate consistently across different OSes (e.g., Ubuntu, RHEL, Windows). For example, the `ansible.builtin.package` module (Figure 2, line 2) serves as a generic package manager that automatically delegates to the appropriate underlying package manager (e.g., `apt`, `dnf`, etc.) based on the target system’s OS. This design helps system administrators to effortlessly deploy their programs across heterogeneous environments. However, a recent study shows this portability promise is often violated. In particular, *OS compatibility bugs*, which are defects that appear only on specific operating systems or versions account for 23% of all IaC bugs [23].

To illustrate OS compatibility bugs, Figure 4 highlights an issue in the `ansible.builtin.user` module [39], which manages user accounts in Unix-like OSes [57]. The test case validates the module’s functionality to migrate a user’s home directory. The test case first sets up the initial system state by creating a user named `movefuser` with a non-empty home directory (lines 1–12). Then, it invokes the module under test with the argument `move_home: true` (lines 20–24). Finally, validation tasks confirm whether the home directory of `movefuser` has been migrated (lines 26–36).

While this test case passes on mainstream Linux distributions (Debian, Fedora, Ubuntu, RHEL), it fails on Alpine Linux. There, the test assertion at line 33 fails because `ansible.builtin.user` reports success without actually moving the directory. The root cause is that Alpine relies on BusyBox utilities, whose `adduser` command lacks support for migrating home directories, unlike the `usermod -m` argument used by other distributions. Since the module does not guard against this missing feature [6], it silently ignores `move_user` on Alpine and falsely reports **Success**. This defect could only be revealed by testing both the module’s reported status (**Success** or **Failure**; Figure 3, lines 27–30) and the actual system state to verify that the home directory was moved. We detected this bug by generating such a test case, reported it to the Ansible developers, and they subsequently fixed it by implementing the missing move functionality.

2.2 Problem Formulation

In this section, we formally introduce the problem that our work tackles.

Definition 2.1 (Ansible Module Semantics). An Ansible module is modeled as a transition function

$$M : \text{State} \times \mathcal{P}(\text{Predicate}) \longrightarrow \text{Status} \times \text{State}$$

where

- $S \in \text{State}$ is a snapshot of all observable, mutable aspects of an operating system, including filesystem contents, installed packages, and running services.
- $\phi \in \text{Predicate}$ is a statement that declaratively describes an abstract system property, e.g., “sqlite3 is installed” or “the nginx service is running.”
- $r \in \text{Status}$ denotes the status of the module invocation, such as **Success** or **Failure**.

Given an initial system state $S \in \text{State}$ and a set of predicates Φ , a module invocation $M(S, \Phi)$ yields a pair of the form $\langle r, S' \rangle$ with the following semantics: If $r = \text{Success}$, then the resulting system state S' satisfies Φ . For example, if $\Phi = \{\text{“Node.js is installed”}\}$, then S' is a state where Node.js binaries and libraries are present in the file system. Otherwise, if $r = \text{Failure}$, then $S' = S$.

In addition to Definition 2.1, we introduce two auxiliary functions. The function $\text{Supported}(m)$ gives the set of OS names (e.g., Debian, Alpine) on which module $m \in \text{Module}$ is supported according to its documentation. We also define the function $\text{Deps}(m)$, which returns the dependency predicates $\Phi \in \mathcal{P}(\text{Predicate})$ required for the module m to execute.

Example: Consider the test case in Figure 3 for `community.general.npm`. Here, $m \in \text{Module}$ is `community.general.npm`, and $\text{Deps}(m)$ consists of the predicate “npm installed in bin path”, as stated in the module’s documentation [12]. The test case satisfies this predicate differently on Debian (lines 2–6) and Arch (lines 8–12) systems.

Definition 2.2 (OS compatibility bug). Let a module $m \in \text{Module}$ and target set of predicate $\Phi \in \mathcal{P}(\text{Predicate})$. An OS compatibility bug exists if:

$$\exists \Phi_o \in \mathcal{P}(\text{Predicate}), \exists i, j \in \text{Supported}(M), \exists S_o^i, S_o^j \in \text{State}, i \neq j$$

such that

- (1) $S_o^i \models \text{Deps}(M) \cup \Phi_o \wedge S_o^j \models \text{Deps}(M) \cup \Phi_o$,
- (2) $\langle r^i, S'^i \rangle = M(S_o^i, \Phi)$,
- (3) $\langle r^j, S'^j \rangle = M(S_o^j, \Phi)$,
- (4) $(r^i \neq r^j) \vee (S'^i \models \Phi \wedge S'^j \not\models \Phi) \vee (S'^i \not\models \Phi \wedge S'^j \models \Phi)$.

That is, a module $m \in \text{Module}$ exhibits an OS compatibility bug when, for two operating systems $i, j \in \text{Supported}(m)$ that both satisfy its dependencies (condition ①), execution yields inconsistent results (condition ④), i.e., either different statuses (e.g., success on one OS but failure on the other), or divergent final system states. Specifically, the final state S'^i in OS i may fail to satisfy the predicate Φ while the corresponding S'^j in OS j does so, and vice versa.

Example: Consider again the example of Figure 4. Here, the `user` module requires no dependency predicates, so $\text{Deps}(m) = \emptyset$. The initial system state before invoking `user` is defined based on Φ_o (lines 1–12):

- “a user with name `movefuser` and home `/tmp/oldhome` exists,”
- “a file at `/tmp/oldhome/testfile.txt` with content `old home` exists.”

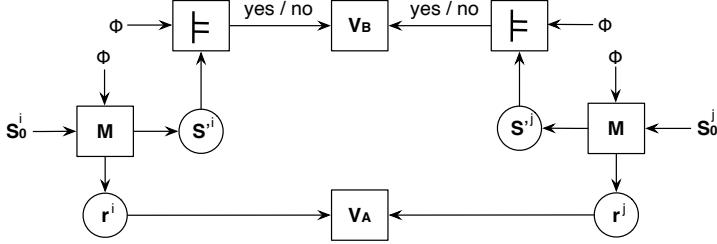


Fig. 5. Cross-platform differential testing in our formal model with two oracle types. V_A checks statuses of module invocations, while V_B checks whether the output state S' satisfies the predicate Φ .

The user module exhibits an OS compatibility bug: although both Debian and Alpine satisfy Φ_0 , invoking the module with $\Phi = \{\text{"the home directory of movefuser is } /tmp/newhome\}$ (lines 20–24) produces different results. On Debian Φ holds after module invocation, but on Alpine it does not, causing the assertion at line 33 to fail.

Relation of the formal model to integration test workflow: Our formal model corresponds to the integration test workflow introduced earlier. Consider a module m executed on operating system i . First, the global setup tasks (Figure 3, lines 1–12) establish a state S_g^i such that $S_g^i \models Deps(m)$. Next, the test case setup tasks (Figure 3, lines 14–18) create a state S_o^i where $S_o^i \models Deps(m) \cup \Phi_0$. The module invocation task then applies M to produce $M(S_o^i, \Phi) = \langle r, S'^i \rangle$. Finally, validation tasks use two types of oracles: validating the module invocation status (Figure 3, lines 27–30), or verifying that S'^i satisfies Φ (Figure 4, lines 27–36).

Remark: While our definition focuses on OS compatibility bugs, the model also captures generic module bugs (i.e., when $S' \not\models \Phi$, or when r does not match the expected **Success** or **Failure** value). We implicitly leverage this in our evaluation (Section 5.1) to uncover OS-agnostic bugs. Future work can apply the model to design oracles for any type of module defect.

Challenges: This paper aims to uncover OS compatibility bugs in popular Ansible modules. Based on Definition 2.2, we need to generate tests that invoke a module with different initial system states and predicates, and then differentially compare validation outcomes across OSes using two oracles using two oracles (Figure 2.2): V_A and V_B . Oracle V_A checks whether the outcome of a module invocation differs across OSes (e.g., failure on one OS but success on another). Oracle V_B evaluates the state-based assertions ($S \models \Phi$ across OSes) and verifies that the predicates hold consistently. Implementing this procedure raises three key technical challenges:

- C1 *Initial system state exploration:* Exploring diverse entry states S_o^i for each OS i , while ensuring that these states satisfy $Deps(m)$.
- C2 *Predicate generation and translation:* Exploring diverse predicates Φ to be provided to the transition function M corresponding to the module under test. Each generated predicate must be translated into the corresponding named arguments used to invoke the Ansible module. For example, when testing the user module of Figure 4, the predicate $\Phi = \{\text{"the home directory of movefuser is } /tmp/newhome\}$ is translated to the named arguments shown in lines 22–24. This translation process is not trivial.
- C3 *State validation:* Designing state-based checks of the form $S' \models \Phi$, which are used by oracle V_B in Figure 5. These state-based checks aim to validate that the final system state S' satisfies a certain predicate set Φ according to $S' \models \Phi$. However, the challenge here lies in how the operator \models is implemented so that it can reliably determine if the given state fulfills the specified predicates.

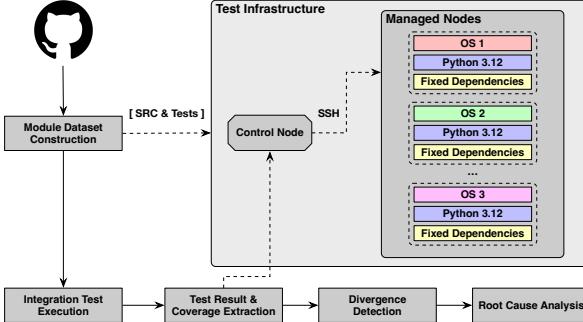


Fig. 6. Empirical study methodology.

Table 1. Tested Linux distributions.

Distribution	Version(s)
Ubuntu	22.04, 24.04
Debian	11, 12
Fedora	40, 41
Rocky Linux	9.3
RHEL UBI	9
openSUSE	Leap 15.6, Tumbleweed
Arch Linux	2025-04-02 (snapshot)
Alpine Linux	3.20.6, 3.21.3

3 Understanding OS-sensitive Integration Test Failures

To better understand the limitations of current integration tests in Ansible modules, we perform a small empirical study. Our goal is to identify how existing tests fail and gain insights for designing new integration tests that can uncover OS compatibility bugs.

3.1 Methodology

Module selection: We query GitHub’s API for non-archived repositories in the `ansible-collections` GitHub organization, which contains all collections in a full Ansible installation [9]. This gives us 118 upstream and community-maintained collections. We also include the core `ansible` repo, which hosts the `ansible.builtin` collection. For each collection, we download the source and identify candidate modules by searching the `plugins/modules` directory, where Ansible module source code is stored [4]. This yields 3,871 candidate modules.

Collections use a standard structure where each module has a corresponding test directory. Matching modules to these directories yields 1,596 modules with dedicated integration tests. For each module, we run the integration test suite in a maintainer-defined Docker container via the `ansible-test integration -docker default` command. For 707 modules, the test suite is skipped due to missing cloud (e.g., AWS, GCP) or vendor-specific (e.g., IBM) credentials. For 455 modules, the test suite fails due to ad-hoc testing techniques (non-standard test placement, alternative execution methods) or requirements for specific hardware (e.g., the `eos_acl_interfaces` module requires an EOS network device). This leaves us with 434 modules for which all integration tests pass in their *default container image*.

OS selection: We select a diverse set of Linux distributions to evaluate the portability of integration test suites across OS families and versions. We source OS images from the official Docker repositories. Table 1 lists the 13 OS / version combinations used in our experiment.

Experimental Setup: Figure 6 shows our empirical study architecture. The *control node* executes Ansible 2.18 and Python 3.12. For each of the 434 modules in our dataset and each of the 13 OS / version combinations our framework: (a) spawns a *managed node container* running the specified OS version, (b) connects via SSH and runs the module’s full integration test suite through the `ansible-test` command, (c) collects the execution result (**Success / Failure**), and (d) gathers coverage metrics for the module’s Python source code. When integration tests fail, we also collect information on the last executed task, allowing us to identify in which phase (setup, module invocation, or validation) the failure occurred (Section 2).

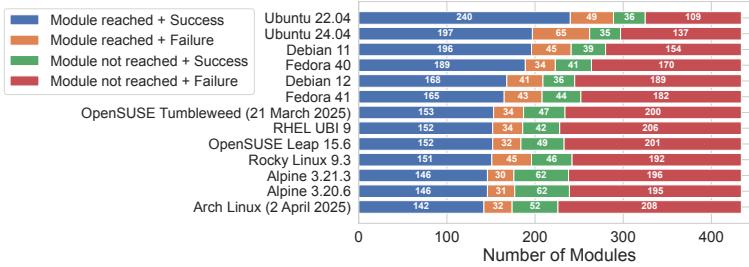


Fig. 7. Integration test execution results across tested operating systems. Each bar categorizes all 434 tested modules into four groups, based on whether their integration tests reached module logic and whether the test suite completed successfully. Green bars indicate that the test suite skipped execution for unsupported OSes. All modules succeeded on at least one OS.

3.2 General Findings

Finding 1: *Across tested distributions, 82% (2,339 / 2,854) of module test failures occur during the setup phases of integration tests.* Most integration tests fail during global or test case setup tasks (recall Section 2), before reaching the module logic. The red bars in Figure 7 illustrate this pattern. While Ansible modules may support various operating systems, their integration tests only cover a limited subset because test environments cannot always be set up properly. Our discussions with module maintainers [26] confirm this: coverage across operating systems is constrained by continuous integration CI resources (limited worker capacity, runtime costs) and technical feasibility, since adding tests for all supported platforms can be tedious.

This highlights the challenge of properly setting up test execution environments (Challenge C1, Section 2.2).

Finding 2: *The code coverage of modules (mean 80%) exhibits small variation (max 0.73) across OSes.* Across the 113 modules that run successfully on all examined OSes, average code coverage ranges from 79.4% (Rocky Linux 9.3) to 80.13% (Ubuntu 22.04). While this might suggest these modules are portable across OSes, prior work [23] shows that 23% of IaC bugs are compatibility-related. A more likely explanation is that OS-specific behaviors reside in the untested 20% of code paths. Reaching these segments requires test cases tailored to specific OSes, which likely explains their absence from current test suites.

Finding 3: *Negative test cases (mean 12.5% per module) are not widely used.* Integration testing must verify that modules fail safely on invalid inputs or improper system states, rather than only exercising the happy path. Prior work [23] reports that 14% of module-level bugs arise from improper handling of such cases, e.g., declaring a file permission with the invalid octal code 0999, or failing to gracefully handle missing dependencies. Table 2 summarizes the prevalence of negative test cases across our studied modules. Notably, 45.4% of modules do not include any negative test cases. These results emphasize the need to account for invalid states and inputs during predicate and system generation (Challenges C1 and C2, Section 2.2).

Finding 4: *State-based assertions (mean 10.8% per module) are not widely used.* Oracles can be either status-based or state-based (recall oracles V_A and V_B in Figure 5). State-based oracles are crucial because they can reveal defects where a module reports **Success** but fails to achieve the desired system state. Table 2 shows that more than half (58.3%) of integration test suites contain no state-based assertions. Interestingly, 5/434 modules, such as

Table 2. Per-module prevalence (percent) of negative test cases and state-based assertions.

Description	5%	Mean	Median	95%	Histogram
Negative-based tests	0.0	12.5	3.9	50.0	
State-based assertions	0.0	10.8	0.0	47.6	

<pre> module: xml short_description: A CRUD-like interface to managing bits of XML files. requirements: - lxml >= 2.3.0 [...] </pre>	<pre> options: file: desc: Path to the file to operate on. type: string xpath: desc: A valid XPath expression. type: string print_match: desc: Print matches. Requires 'xpath' to also be set. type: boolean [...] </pre>	EXAMPLES: <pre> community.general.xml: path: /foo/bar.xml xpath: /business/rating print_match: true [...] </pre> RETURN: <pre> count: desc: The count of xpath matches. type: integer sample: 2 [...] </pre>
---	---	--

(a) Description and dependencies.

(b) Options (parameters).

(c) Examples and return values.

Fig. 9. Upstream YAML documentation for the `community.general.xml` [7] module.

`ansible.builtin.iptables`[5], rely exclusively on state-based assertions, suggesting that maintainers recognize their value. These findings highlight the need for broader adoption of state-based assertions (Challenge C3, Section 2.2) to uncover bugs that status checks alone would miss.

4 Design of cROSSIBLE

Overview: Building upon our formal model and empirical findings, we present cROSSIBLE, a framework that *automatically generates integration tests to detect OS compatibility issues* in Ansible modules. cROSSIBLE leverages the official documentation of Ansible modules and large language models (LLMs) to tackle the challenges of finding OS compatibility bugs.

Figure 8 shows the architecture. Given an input module, cROSSIBLE first extracts relevant documentation metadata (e.g., dependencies, input parameters). This metadata is passed to an LLM, which generates integration test scenarios in natural language to exercise OS-specific module behaviors. In a second stage, these scenarios are converted into concrete YAML tasks conforming to the Ansible DSL (*Ansible DSL generation*). The generated tasks are validated for syntactic and semantic correctness using automated Ansible tooling. If validation fails, the framework re-prompts the LLM with specific error messages to iteratively repair the YAML (*DSL repair*). Each validated integration test suite is executed across multiple Linux distributions using our cross-platform testing infrastructure (recall Figure 6). The execution traces are compared to detect OS compatibility bugs. Finally, cROSSIBLE employs deterministic filtering (*false positive filtering*) to exclude divergences occurring in setup phases, which do not represent genuine module bugs.

4.1 Design Principles

Employing Large Language Models: Our study shows that even expert-written integration tests are fragile to system state and execution semantics (Section 3.2), making it difficult to anticipate all OS-induced failures and preconditions. Encoding OS-specific quirks, DSL constraints, and system-state requirements in fixed templates is challenging. Core Ansible maintainers [26] confirm that

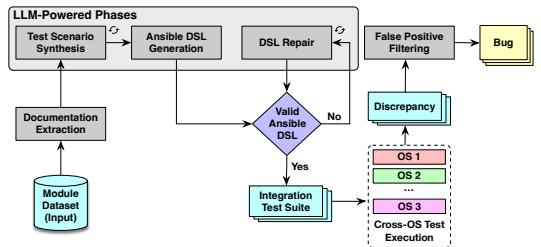


Fig. 8. Architecture of cROSSIBLE.

the main barrier to thorough integration testing is the substantial human effort required to write and maintain tests.

Large language models (LLMs) offer a promising alternative. Prior work [32, 34, 61] shows that LLMs can generate valid IaC code from structured or natural-language descriptions and detect defects in DSL [16]. Recent work [11, 38] also demonstrates that LLMs effectively detect misconfigurations in declarative configuration files, analogous to declarative module invocations in our setting. We therefore design an LLM-assisted testing framework guided by our empirical findings.

API-guided Synthesis: A key challenge of testing Ansible modules is their diversity. There are modules for managing various resources: databases, networks, file systems, and more. Designing a system or predicate generator capable of producing inputs for all these diverse domains is challenging. To address this, our approach targets a specific Ansible module and provides its documentation directly to the LLM. Each module’s source contains structured YAML documentation that guides each part of the integration test (setup tasks, module invocation, etc.). Figure 9 shows the documentation for the `community.general.xml` module:

1. *Description and external dependencies.* Figure 9a illustrates a brief description of the module’s purpose and a list of external dependencies. Notably, this part of the documentation specifies the premises, the predicates that must hold for the module’s dependencies in order for it to operate successfully (i.e., $Deps(m)$, Section 2.2). For example, `community.general.xml` requires “the Python package `lxml > 2.3.0` is installed”, while `community.general.npm` requires “`npm` installed in bin path” [12]. Our approach leverages these predicate to produce a state that satisfies them via an LLM. This addresses Challenge C1 (Section 2.2): ensuring $S_0^i \models D deps(m)$ on the OS i .

2. *Parameter semantics and input types.* A list of parameters with their semantics and input types (e.g., Figure 9b). This information enables the LLM to generate suitable predicates for module invocation and validation (e.g., “the attribute `foo` is absent from the XML file at `/bar.xml`”) and translate them into valid Ansible DSL syntax. This addresses Challenge C2 (Section 2.2) by producing diverse predicates Φ , and informs oracle design (Challenge C3) by establishing expected behavior.

3. *Examples and return values.* Example invocations of the module and the semantics of their return values (e.g., Figure 9c). This further helps the LLM generate proper module invocations and validation tasks. Our API-guided synthesis approach aligns with recent work showing that LLMs generate IaC programs most effectively when combined with retrieval-augmented generation (RAG) [34].

Self-planning code generation: Our design follows the “self-planning code generation” paradigm [33], which introduces an explicit planning stage prior to code generation. In this paradigm, the model first generates a structured plan in natural language that captures the high-level logic of the desired integration test (*test scenario synthesis*, Figure 8). The test plan is then translated into executable Ansible DSL code (*Ansible DSL generation*). This two-phase approach enables cROSSIBLE to explore richer and more precise test ideas without considering Ansible’s DSL constraints.

4.2 Architecture

Step 1: Documentation extraction: The input to cROSSIBLE is an Ansible module m . Each module’s source code embeds its documentation and usage examples as top-level string variables: DOCUMENTATION, RETURN, and EXAMPLES. cROSSIBLE statically retrieves this documentation (Figure 9) by parsing the module’s abstract syntax tree (using Python’s `ast` module). This step ensures that cROSSIBLE captures the exact documentation version corresponding to the code under test.

Step 2: Test scenario synthesis: cROSSIBLE uses an LLM to generate structured integration test scenarios in natural language based on the extracted documentation. Figure 10 shows a simplified

[System prompt]: You are an expert in Ansible, Linux OS internals, and integration testing. Your task is to uncover compatibility bugs across diverse Linux distributions by designing documentation-compliant integration test scenarios.

[User prompt]: Generate exactly 10 integration test scenarios for the module `{module_name}` from collection `{collection}`. The goal is to identify OS compatibility issues across multiple Linux distributions.

- Documentation:
 - `{description}`
 - `{examples}`
 - `{return_output}`
- Execution Context:
 - Ansible 2.18, Python 3.12, minimal Docker base images, OS list: `{os_list}`
- Global Setup Block:
 - Install all required packages and dependencies (explicitly from Documentation provided).
 - Enable required services (e.g., databases, daemons, systemd).
 - Create necessary users, directories, config files.
 - Assume no pre-installed tools. Redundant installations are allowed for reliability.
 - Use shell commands when necessary to avoid undocumented Ansible features.
- Scenario Requirements (10 total):
 - Each scenario must include:
 - name: short, unique identifier
 - goal: why this scenario may reveal OS compatibility issues
 - setup: preparatory steps for system state
 - main_task: exact module invocation with parameters from Documentation
 - oracle: assertions validating outcome
 - negative_test: yes/no
- Constraints:
 - At least 5 scenarios must be negative tests (expected to fail, but may succeed on specific platforms)
 - The oracle must also include tasks explicitly examining system state to validate outcome.
 - Do not return YAML or explanations, but only structured plain text.

[User prompt]:
 You are given a batch of test scenarios for the module `{collection}.{module_name}`.
 Each scenario includes name, goal, setup, main_task (with parameters), oracle (assert-style conditions), negative_test (yes/no)

Your task: Convert this into valid Ansible YAML for integration tests.

Output Instructions:

1. If present, begin with the **Global Setup Tasks** block.
2. For each scenario, output:
 - Setup tasks
 - Main task (register: result; use ignore_errors: true for negative tests)
 - Oracle: each condition as a separate `ansible.builtin.assert` task

Assertion Guidance:

- Each oracle condition must be implemented as its own `ansible.builtin.assert` task.
- Use only valid fields: `result.failed`, `result.changed`, `result.ok`.

OS-specific Instructions:

- Scenarios run across OSes: "Alpine", "Archlinux", "Debian", "Fedora", "SUSE", "Rocky", "RedHat", "Ubuntu".
- If setup differs per OS, use conditionals:
 For example:
`when: ansible_facts["distribution"] == "Ubuntu"`
 or
`when: ansible_facts["distribution"] in ["RedHat", "Fedora"]`

Allowed Modules (for setup, oracle):
`ansible.builtin.file`,
`ansible.builtin.package`,
`ansible.builtin.pip`
 Avoid undocumented or invalid modules.
 Use `ansible.builtin.shell` as a safe fallback.

YAML Quoting Reminder:
 Always quote names/values that contain special characters (e.g., colons, pipes, angle brackets).

Fig. 10. Prompt template for *test scenario synthesis*.Fig. 11. Prompt template for *Ansible DSL generation*.

version of the prompt. Each scenario follows the integration test workflow described in Section 2. A scenario includes a name and goal stating why it may expose OS compatibility issues. It specifies predicates that define the execution environment (e.g., installing dependencies, refining system state) and describes the module invocation using parameters from the documentation, along with strategies to validate the resulting system state.

The prompt emphasizes properly setting up the test execution environment by instructing the LLM to retrieve dependencies from documentation, assume nothing is pre-installed, and enable required services. This reduces setup failures (Finding 1, Section 3.2). The prompt also requests both positive scenarios (expecting successful execution) and negative scenarios that test module behavior under ill-formed inputs and states, addressing the lack of such tests in Finding 3. Finally, it directs the LLM to validate the resulting state to ensure module correctness (Finding 4).

Example: Figure 12a shows the test scenario synthesis result for the `ansible.builtin.user` module. The scenario involves creating a user and then validating that the user's home directory is moved to a new location. This test scenario is later translated into an Ansible integration test (Figure 4) by a subsequent step of cROSSIBLE.

Step 3: Ansible DSL Generation: The structured test scenarios from the previous stage are converted into executable Ansible integration tests. This is done through a prompt (Figure 11) that instructs the LLM to generate an Ansible playbook in YAML format. Note that the prompt provides

```

name: MoveHomePreservesFiles
goal: Verify `move_home` true moves existing home
      contents and updates file entry

setup:
- user: name=movefuser home=/tmp/oldhome
  create_home=yes state=present
- copy: dest=/tmp/oldhome/testfile.txt
  content="old home"

main_task:
user:
  name: movefuser
  home: /tmp/newhome
  move_home: true
  state: present

oracle:
- assert: user_result is succeeded
- stat: /tmp/newhome/testfile.txt -> exists
negative_test: no

```

(a) Test scenario

```

TASK [Create user with home /tmp/oldhome]
changed: [localhost]

TASK [Create file in the original home]
changed: [localhost]

TASK [Move home directory]
changed: [localhost]

TASK [Stat file in new home]
ok: [localhost]

TASK [Ensure home moved]
fatal: [localhost]: FAILED! => {
  "assertion": "movedfile.stat.exists"
  "changed": false,
  "evaluated_to": false,
  "msg": "Assertion failed"
}

PLAY RECAP
localhost : ok=4  changed=3  failed=1

```

(b) Execution trace

Fig. 12. The test scenario and the execution trace of the integration test presented in Figure 4.

guidance on OS-specific conditionals, including concrete examples and the complete list of OS distribution strings (e.g., `ansible_facts["distribution"] == "Ubuntu"`). These conditionals are used only when setup tasks require OS-specific handling, preventing failures from incorrect platform-specific DSL usage.

Example: This step converts the test scenario for the `ansible.builtin.user` module (Figure 12a) into the Ansible integration test of Figure 4, which we discussed in Section 2.1.

Step 4: DSL Validation and Repair: Each batch of ten generated integration tests undergoes lightweight validation before execution. The validation ensures that tests (1) are well-formed (no YAML syntax errors) and (2) respect Ansible semantics, such as correct variable references and properly structured control constructs (e.g., `when` clauses, loops). This follows a fail-fast strategy: each test is validated and stopped on the first error. Tests that pass validation proceed to execution, while those that fail have their errors recorded and are sent to the DSL repair stage.

Batches rejected during validation enter the DSL Repair phase, where the LLM corrects reported issues. This follows prior work showing that feedback-guided prompting improves IaC code generation [61]. For each failing batch, the model receives the full YAML content, validator output, and exact error location. It is instructed to apply minimal fixes while preserving task logic. Because validation uses a fail-fast strategy, multiple repair-validation cycles may be needed to resolve all errors (see Section 5.4 for details).

Step 5: Cross-OS execution and divergence detection: We considered two execution strategies for the generated integration tests. In terms of our formal model (Section 2.2), both execute $M(S_o^i, \Phi)$ across every supported OS i but differ in how they establish the required global state ($S_o^i \models Deps(M) \cup \Phi_o$). The first strategy runs generated tests independently, while the second appends them to the module's existing integration tests, reusing the maintainers' setup tasks. Across a sample of 40 modules with passing tests on all OSes, the second approach enabled 20% more generated tests to complete without setup failures. We therefore adopt the second strategy, limiting evaluation to modules whose existing integration tests pass on at least two OSes.

When divergences occur across OSes, we retain only cases where one outcome is failure and another is success, as these divergences more likely indicate genuine OS compatibility issues.

Step 6: False positive filtering: Divergences occurring before module invocation are not genuine compatibility bugs. Instead, they indicate faulty test setup that fails to establish S_o correctly. Our

synthesized scenarios contain annotations indicating each task’s phase (e.g., “setup”, “main_task”, “oracle” in Figure 12a). DSL generation preserves these annotations as comments (e.g., Figure 4, lines 1, 14, 19, 26).

Our filtering technique takes as input the playbook execution trace of the failing test (e.g., Figure 12b) and the YAML source of the test case. We traverse the tasks to locate the failing one; if it precedes module invocation, we classify the divergence as a false positive and exclude it from manual analysis (Section 5.4).

4.3 Implementation

cROSSIBLE consists of 2.5k lines of Python code. Its input is an Ansible module’s source code and integration tests. The generated tests undergo two validation steps: syntactic checks using the `ruamel.yaml` library and semantic checking using the `ansible-playbook -syntax-check` command. The semantic check uses the same method as Ansible’s own tests. For cross-OS execution, we reuse the container-based infrastructure from our empirical study (Section 3.1). To append generated tests to existing ones, we identify the module’s integration test entry point (typically `main.yml`) and add generated tests at the end.

Prompt engineering: Table 3 lists the LLM variants and temperatures used in each phase of cROSSIBLE. cROSSIBLE employs the DeepSeek family models [24, 25] for their cost-performance balance and prior success in generating Ansible programs [31]. We use these models via their official APIs.

For test scenario synthesis, we use the reasoning variant (R1-0528) as this phase requires creativity and explicit reasoning. For Ansible DSL generation and repair, we use the core model (V3-0324) as these steps require precision and syntax compliance. Sampling temperature strongly affects output quality [31]. Higher values (0.6–0.8) yield diverse, creative results, while lower ones (0.2–0.4) improve determinism. For the test scenario synthesis step the temperature score is 0.6, favoring diversity to uncover OS-sensitive edge cases, consistent with work [59]. For Ansible DSL generation and repair, our temperature score is 0.0, prioritizing precision and determinism.

5 Evaluation

We evaluate cROSSIBLE based on the following research questions:

RQ1 How effective is cROSSIBLE in finding OS compatibility bugs? (Section 5.1)

RQ2 Which OSes are affected most, and what are the symptoms and root causes of found bugs? (Section 5.2)

RQ3 To what extent does cROSSIBLE improve module code coverage compared to the original integration test suites? (Section 5.3)

RQ4 How capable is cROSSIBLE at producing valid tests and what is the prevalence of false positives? (Section 5.4)

Experimental Setup: We reuse the experimental setup from our empirical study (Section 3): the control node runs Python 3.12 and Ansible Core 2.18, while managed nodes run the OSes listed in Table 1. We also reuse the 434-module dataset, applying one additional filter. Since our strategy appends generated tests to existing integration tests, we keep only modules whose original tests pass on at least two OSes for differential testing. This yields 259 modules for evaluation. cROSSIBLE uses the official DeepSeek API for all LLM phases. For each module, cROSSIBLE generates 18 batches

Table 3. Model/temperature used for each phase.

Phase	Model	Temp.
Test Scenario Synthesis	DeepSeek-R1-0528	0.6
DSL Generation	DeepSeek-V3-0324	0.0
DSL Repair	DeepSeek-V3-0324	0.0

Table 4. (a) Distribution of reported bugs according to resolution status and OS sensitivity, (b) number of affected operating systems for OS-sensitive bugs, (c) bug count by Ansible collection.

Status	compat. bug	OS-Agnostic	Total	OS family	Bug count	Collection	Count
Reported	5	3	8	Alpine	8	ansible.builtin	13
Confirmed	11	5	16	Debian-based	7	community.general	13
Fixed	5	6	11	RedHat-based	5	ansible.posix	4
Wontfix	1	0	1	OpenSUSE	1	community.dns	2
Total	22	14	36	All \ Debian-based	1	Others	4
(a)		(b)		Total	22	(c)	

of 10 scenarios (180 tests per module). With 259 modules total, this yields 46,620 test scenarios overall. The full workload runs for 12 hours with a total API cost of \$42.

5.1 RQ1: Effectiveness of cROSSIBLE

When a test fails on every OS, cROSSIBLE logs a warning message, along with crash markers (e.g., Python tracebacks). Although these issues fall outside our primary focus on OS portability, they represent genuine module bugs. We report them alongside OS compatibility bugs, noting that their detection is a by-product of the cROSSIBLE pipeline.

Table 4a summarizes the bug-finding results of cROSSIBLE. Within 12 hours, cROSSIBLE uncovers 36 previously unknown bugs, including 22 OS compatibility issues, with 27 confirmed and 11 fixed at the time of writing. Table 4c shows that 13 bugs affect the core `ansible.builtin` collection, which resides in the main Ansible repository and is maintained by core developers. All of them are OS-sensitive bugs, and all of them have been confirmed. An equal number affects `community.general`, the most popular third-party collection (most GitHub stars/issues). In total, the detected bugs span 25 unique modules within the collections listed in Table 4c, representing 10% of the 259 modules tested. These modules span diverse functionality: package managers, cryptographic secrets, OS resources (e.g., users, crontabs), and XML files.

Of the 36 detected bugs, 22 are OS compatibility issues while 14 are OS-agnostic, failing across all examined OSes. This highlights the utility of our approach for uncovering both portability and general bugs in Ansible modules.

5.2 RQ2: Qualitative Bug Characteristics

Affected operating systems: Table 4b shows the distribution of OS compatibility bugs. Alpine is most affected (8 bugs), as modules frequently assume that glibc or GNU tools are present, while Alpine relies on musl [41] and BusyBox [18], which are not always compatible. Fixes for such bugs commonly add conditional logic to handle Alpine-specific behavior. Despite this overhead, Alpine support is essential due to its wide container adoption. On DockerHub, Alpine has 12.6M weekly downloads [19], compared to 4.5M for Ubuntu [21] and 4.0M for Debian [20].

We also observe that bugs affecting one OS family member affect all others: cROSSIBLE found seven bugs in Debian-based systems and five in RHEL-based systems. This suggests that testing one distribution per family may suffice, especially given limited CI resources.

Bug symptoms: Figure 13a shows the distribution of symptoms grouped by OS sensitivity. We reuse the taxonomy of Drosos et al. [23], adding a new category: *soundness* issues.

The most common symptom is *Internal error (crash)*, accounting for more than half (19/36) of bugs found. For example, a fixed bug in `ansible.builtin.service_facts` caused a `KeyError` on Alpine Linux while working correctly on Ubuntu. Alpine uses OpenRC [45] as its `init` system, and although the module explicitly supports OpenRC, a faulty regular expression caused the `KeyError`.

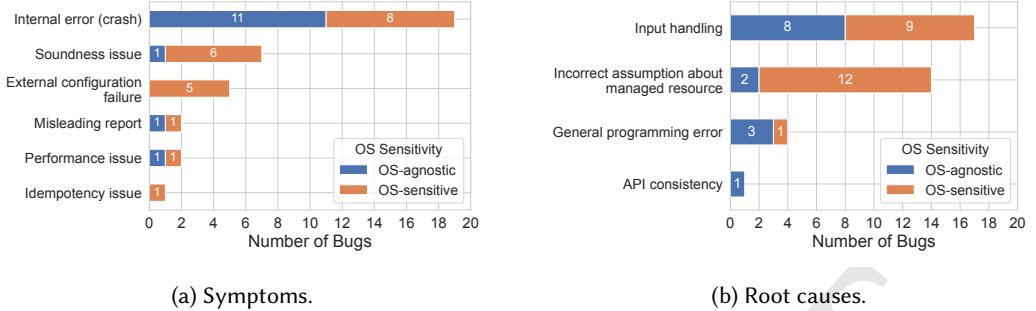


Fig. 13. Qualitative results for the 36 found bugs.

Soundness issues represent cases where a module’s status value (ok, changed, failed) does not reflect the true system state. Only state-based oracles can detect such issues (Figure 5). For example, cROSSIBLE uncovered a bug in `community.general.apk` where invoking the module with an empty package name returns changed while doing nothing. The `ansible.builtin.user` module bug (Figure 4) is another soundness issue.

External configuration failure occurs when modules terminate gracefully (e.g., through `try/except`) upon encountering failures from external commands (e.g., Figure 1). We encountered this only in OS-sensitive bugs, highlighting the need for modules to account for implementation differences in equivalent resources (e.g., `cron`) to ensure portability.

Root causes: We manually investigate each bug’s root cause by inspecting failing test logs, module source code, maintainer discussions, and fixes. We identify four categories, with their distribution shown in Figure 13b.

The most common root cause is improper *input handling*, where modules insufficiently validate input parameters, accounting for over half (17/36) of bugs. Most are uncovered through negative test cases where cROSSIBLE provides invalid inputs expecting failure.

To provide a uniform, portable API, Ansible modules often wrap managed resources such as package managers (e.g., `apt`, `dnf`) and services (e.g., `cron`), assuming consistent behavior and inferring state from outputs and exit codes. However, forked or alternative implementations (e.g., `cronie` vs. `vixie cron` in Figure 1) can violate these assumptions, accounting for over half (12/22) of OS-sensitive bugs. For example, the `community.general.gem` module misinterpreted a zero exit code as successful uninstall, even though the targeted gem was not removed on Ubuntu. The fix explicitly verified post-state rather than assuming semantics from exit codes.

Finally, cROSSIBLE finds one bug caused by an *inconsistency* between documented and actual API. While documentation for `community.dns.nameserver_record_info` claims to support the “ALL” parameter, this actually raised a `KeyError`.

5.3 RQ3: Code Coverage

Table 5 summarizes code coverage scores across modules for each OS. We measure (1) baseline coverage (cov_0) from the original integration suite, (2) augmented coverage (cov_a) after adding cROSSIBLE’s tests, and (3) the per-module change (Δcov). Note that Δcov reflects individual module improvements, not aggregated totals.

Across all OSes, average code coverage rises from 63.2% to 75.5%, while medians improve from 78% to 85%. In one extreme case, a single module experiences 95% higher code coverage using cROSSIBLE’s tests.

Table 5. Module coverage metrics: baseline coverage (cov_0) from the unmodified integration test suite, augmented coverage (cov_a) after adding our generated tests, and coverage change ($\Delta cov = \text{augmented minus baseline}$).

OS	Median			Mean			Std		
	cov_0	cov_a	Δcov	cov_0	cov_a	Δcov	cov_0	cov_a	Δcov
Ubuntu (22.04)	81%	86%	3%	68.8%	78.1%	9.3%	29.8%	21.8%	15.6%
Ubuntu (24.04)	80.5%	86%	3%	68.9%	78.4%	9.4%	29.5%	21.6%	16%
Debian (11)	78%	84%	3%	65.1%	75.7%	10.6%	31.5%	23.2%	17.7%
Debian (12)	79%	85%	4%	65%	76.4%	11.3%	31.8%	23.5%	18%
RHEL (9)	78%	85%	6%	63.8%	75.8%	12%	32.5%	23.8%	18.1%
Rocky Linux (9.3)	75%	84%	5%	59.4%	72.7%	13.2%	34%	25.9%	21.2%
Alpine (3.20.6)	76%	85%	7%	57.4%	73.2%	15.8%	36.6%	25.9%	22.2%
Alpine (3.21.3)	76.3%	85%	6.5%	57.7%	74.1%	16.4%	36.6%	25.2%	23.2%
Fedora (40)	80%	86%	4%	68%	79.1%	11%	30.8%	21.1%	17.7%
Fedora (41)	78%	85%	4%	62.7%	74.4%	11.6%	33%	25%	18.3%
Arch Linux (latest)	76.6%	85%	6%	59.1%	74.1%	15%	35.9%	25.3%	21.5%
OpenSUSE Leap (15.6)	77%	85%	6%	60.6%	74.6%	14%	34.6%	25.1%	20.5%
OpenSUSE Tumbleweed (latest)	77.2%	85%	6%	61.2%	74.4%	13.1%	34.4%	25.2%	19.7%
Total	78%	85%	5%	63.2%	75.5%	12.3%	33.2%	24%	19.2%

Coverage gains vary across OSes. Alpine exhibits the largest increases (15–16% mean, 6–7% median) because it is most frequently skipped in upstream test suites, aligning with our empirical study findings (Figure 7, green bar). By synthesizing OS-aware setup tasks, CROSsible reaches previously untested Alpine paths, explaining both its high coverage gains and large share of OS-specific bugs. Standard deviations of 15–23% indicate that some modules benefit much more than others.

5.4 RQ4: Test Validity and False Positives

Test validity and DSL repair: We evaluate CROSsible’s ability to produce valid test cases. For each module, CROSsible generates 18 batches of 10 tests, totaling 4,662 batches across all tested modules. As shown in Table 6, 79.34% of batches pass validation immediately, 97.06% succeed after one retry, with only one batch requiring seven retries. Passing validation ensures that `ansible-test` can parse and execute the tasks. High success rates benefit from small batch sizes, which keep LLM context short [2] and improve error localization. This demonstrates that LLMs can reliably generate syntactically and semantically valid IaC programs for complex infrastructure scenarios.

Filtering false positives: Execution produced 3,373 divergences. Our false-positive filter reduced this by 86% to 475 task failures. After manual inspection, keeping one representative task when tests fail on multiple OSes, 99 unique tasks remained: 36 genuine bugs (RQ1) and 63 false positives.

False positives arise from two main causes. First, missing setup dependencies surface as module invocation errors (e.g., `ImportError` or `command not found`), violating our compatibility bug definition since ($S_0 \not\models Deps(M)$). This accounts for 43/63 cases, highlighting incomplete module dependency documentation [42]. Second, in 20 cases, the OS cannot satisfy predicate Φ because the functionality is unsupported on that OS. While not portability bugs, these expose documentation gaps since none of these incompatibilities were explicitly noted. For instance,

Table 6. Distribution of required DSL repair iterations across all 4,662 module–batch pairs (18 batches \times 259 modules).

Retries	# Batches	Fraction (%)	Cumulative (%)
0	3,699	79.34	79.34
1	826	17.72	97.06
2	86	1.84	98.91
3	30	0.64	99.55
4	10	0.21	99.76
5	4	0.09	99.85
6	6	0.13	99.98
7	1	0.02	100.00

`community.crypto.openssh_cert` does not support SHA-1 on RedHat, only indirectly implied in its integration tests [17]

6 Related Work

Defects in IaC: The most comprehensive prior study is by Drosos et al. [23], who characterize IaC defects across application (playbook) and runtime (module) layers. They highlight OS sensitivity and the difficulty of reproducing state-dependent bugs. Our results confirm and extend these insights for Ansible by uncovering numerous OS-specific defects in Ansible.

Drosos et al. [23] also identify a gap in techniques for the *runtime* layer (i.e., Ansible modules). Hassan et al. [32] partly address this by studying state-reconciliation defects and using LLMs to generate playbooks from issue reports, but rely solely on crashes as oracles.

cROSSIBLE uses a differential oracle to detect compatibility bugs and checks the resulting system state to uncover soundness bugs (Figure 5). Opdebeeck et al. [42] study Ansible module dependencies and highlight that loosely documenting dependencies is inadequate.

Work at the *application* layer (DSL programs) is largely orthogonal to ours. Static techniques target code smells [43, 51, 53] security risks [15, 44, 49, 50] and semantic violations [47, 52], while dynamic techniques analyze runtime behavior to surface dependency faults [56] or deployment errors [13, 30, 54, 55]. These approaches operate on declarative specifications rather than the underlying runtime we study.

LLMs for IaC: We cover related work on applying LLMs to IaC (e.g., code generation and misconfiguration detection) is covered in the rationale for our first design principle (Section 4.1).

Other related studies: Furthering the scope, related work on Kubernetes operators explores varied entry states S_0 (Sieve [58]) and desired states Φ (Acto [28]) to test reconciliation logic. A key distinction is architectural: Kubernetes operators interact exclusively with a centralized API server [36], the authoritative source of system state, enabling oracles to be implemented via API requests. In contrast, Ansible lacks a single ground-truth API, so cROSSIBLE must infer properties and semantics from arbitrary applications. To address this challenge, cROSSIBLE leverages an LLM with targeted guidance informed by our empirical findings.

7 Conclusion

We presented cROSSIBLE, a practical, empirically guided LLM framework that synthesizes integration tests from Ansible module documentation to stress-test OS portability. Its design builds on a formal model of OS compatibility bugs and a large-scale empirical study exposing weaknesses in upstream integration tests, particularly their lack of sophisticated test oracles.

Our evaluation on hundreds of modules shows that cROSSIBLE effectively finds OS compatibility bugs while significantly increasing code coverage. Beyond bug discovery, cROSSIBLE can be used to enrich existing test suites.

8 Data Availability

We provide the source code of cROSSIBLE together with all scripts and data needed to reproduce our results in the supplementary material. We plan to make cROSSIBLE publicly available, concurrently with this paper’s publication.

References

- [1] Amazon Web Services, Inc. or its affiliates. 2017. *Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region*. <https://aws.amazon.com/message/41926/> [Online; accessed 7-July-2025].
- [2] Chenxin An, Jun Zhang, Ming Zhong, Lei Li, Shanshan Gong, Yao Luo, Jingjing Xu, and Lingpeng Kong. 2025. Why Does the Effective Context Length of LLMs Fall Short?. In *The Thirteenth International Conference on Learning Representations*.

- <https://openreview.net/forum?id=eoIn5WgrPx>
- [3] Ansible. 2024. *Ansible Galaxy*. <https://galaxy.ansible.com/ui/> [Online; accessed 7-July-2025].
 - [4] Ansible. 2024. *Modules*. <https://docs.ansible.com/ansible/latest/plugins/module.html#enabling-modules> [Online; accessed 7-July-2025].
 - [5] Ansible Contributors. 2025. *ansible.builtin.iptables implementation*. https://github.com/ansible/ansible/blob/devel/test/integration/targets/iptables/tasks/chain_management.yml [Online; accessed 31-August-2025].
 - [6] Ansible Core Team. 2025. *ansible.builtin.user implementation*. <https://github.com/ansible/ansible/blob/devel/lib/ansible/modules/user.py#L3109> [Online; accessed 31-August-2025].
 - [7] Ansible Documentation. 2025. *community.general.xml – Manage bits and pieces of XML files or strings*. https://docs.ansible.com/ansible/latest/collections/community/general/xml_module.html [Online; accessed 31-August-2025].
 - [8] Ansible documentation. 2025. *Integration tests*. https://docs.ansible.com/ansible/latest/dev_guide/testing_integration.html [Online; accessed 31-August-2025].
 - [9] Ansible Documentation. 2025. *Selecting an Ansible Package and Version to Install*. [Online; accessed 31-August-2025].
 - [10] Ansible Documentation. 2025. *Testing Ansible and Collections*. https://docs.ansible.com/ansible/latest/dev_guide/testing_running_locally.html [Online; accessed 31-August-2025].
 - [11] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. 2024. Automatic Root Cause Analysis via Large Language Models for Cloud Incidents. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys'24)*.
 - [12] Chris Hoffman. 2025. *community.general.npm module – Manage node.js packages with npm*. https://docs.ansible.com/ansible/latest/collections/community/general/npm_module.html [Online; accessed 31-August-2025].
 - [13] Emilio Coppa, Daniel Sokolowski, and Guido Salvaneschi. 2025. Hybrid Fuzzing of Infrastructure as Code Programs (Short Paper). In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (Clarion Hotel Trondheim, Trondheim, Norway) (ISSTA Companion '25)*. Association for Computing Machinery, New York, NY, USA, 92–97. doi:[10.1145/3713081.3731721](https://doi.org/10.1145/3713081.3731721)
 - [14] crontie maintainers. 2025. *Crontie cron daemon project*. <https://github.com/crontie-cron/crontie> [Online; accessed 31-August-2025].
 - [15] Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. 2020. Automatically detecting risky scripts in infrastructure code. In *Proceedings of the 11th ACM Symposium on Cloud Computing (Virtual Event, USA) (SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 358–371. doi:[10.1145/3419111.3421303](https://doi.org/10.1145/3419111.3421303)
 - [16] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian A. Tamburri. 2022. Within-Project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics. *IEEE Transactions on Software Engineering* 48, 6 (2022), 2086–2104. doi:[10.1109/TSE.2021.3051492](https://doi.org/10.1109/TSE.2021.3051492)
 - [17] David Kainz. 2025. *community.crypto.openssh_cert implementation*. https://github.com/ansible-collections/community.crypto/blob/main/tests/integration/targets/openssh_cert/tests/key_idempotency.yml#L80 [Online; accessed 31-August-2025].
 - [18] Denys Vlasenko. 2025. *BusyBox*. <https://busybox.net/> [Online; accessed 31-August-2025].
 - [19] Docker Library. 2025. Alpine Official Image · Docker Hub. https://hub.docker.com/_/alpine. Accessed: 2025-09-07.
 - [20] Docker Library. 2025. Debian Official Image · Docker Hub. https://hub.docker.com/_/debian. Accessed: 2025-09-07.
 - [21] Docker Library. 2025. Ubuntu Official Image · Docker Hub. https://hub.docker.com/_/ubuntu. Accessed: 2025-09-07.
 - [22] Doug Luce. 2025. *community.general.cronvar module – Manage variables in crontabs*. https://docs.ansible.com/ansible/latest/collections/community/general/cronvar_module.html [Online; accessed 31-August-2025].
 - [23] Georgios-Petros Drosos, Thodoris Sotiropoulos, Georgios Alexopoulos, Dimitris Mitropoulos, and Zhendong Su. 2024. When Your Infrastructure Is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 359 (Oct. 2024), 31 pages. doi:[10.1145/3689799](https://doi.org/10.1145/3689799)
 - [24] DeepSeek-AI *et al.* 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] <https://arxiv.org/abs/2501.12948>
 - [25] DeepSeek-AI *et al.* 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] <https://arxiv.org/abs/2412.19437>
 - [26] Felix Fontein and Alexei Znamensky. 2025. OS Coverage in Ansible Integration Tests. Private correspondence. Email, 5 Sep 2025.
 - [27] GitHub, Inc. 2014. *DNS Outage Post Mortem*. <https://github.blog/2014-01-18-dns-outage-post-mortem/> [Online; accessed 7-July-2025].
 - [28] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunson, and Tianyin Xu. 2023. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th Symposium on Operating Systems Principles (Koblenz, Germany) (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 96–112. doi:[10.1145/3600006.3613161](https://doi.org/10.1145/3600006.3613161)

- [29] Michele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. 2019. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSM)*. 580–589. doi:[10.1109/ICSM500092](https://doi.org/10.1109/ICSM500092)
- [30] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. 2016. Asserting reliable convergence for configuration management scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) (*OOPSLA 2016*). Association for Computing Machinery, New York, NY, USA, 328–343. doi:[10.1145/2983990.2984000](https://doi.org/10.1145/2983990.2984000)
- [31] Md Mahadi Hassan, John Salvador, Akond Rahman, and Santu Karmaker. 2025. Large Language Models for IT Automation Tasks: Are We There Yet? arXiv:2505.20505 [cs.CL] <https://arxiv.org/abs/2505.20505>
- [32] Md Mahadi Hassan, John Salvador, Shubhra Kanti Karmaker Santu, and Akond Rahman. 2024. State Reconciliation Defects in Infrastructure as Code. *Proc. ACM Softw. Eng.* 1, FSE, Article 83 (jul 2024), 24 pages. doi:[10.1145/3660790](https://doi.org/10.1145/3660790)
- [33] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-Planning Code Generation with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 33, 7, Article 182 (Sept. 2024), 30 pages. doi:[10.1145/3672456](https://doi.org/10.1145/3672456)
- [34] Patrick Tser Jern Kon, Jiachen Liu, Yiming Qiu, Weijun Fan, Ting He, Lei Lin, Haoran Zhang, Owen M. Park, George S. Elengikal, Yuxin Kang, Ang Chen, Mosharaf Chowdhury, Myungjin Lee, and Xinyu Wang. 2024. IaC-Eval: A Code Generation Benchmark for Cloud Infrastructure-as-Code Programs. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 134488–134506. https://proceedings.neurips.cc/paper_files/paper/2024/file/f26b29298ae8acd94bd7e839688e329b-Paper-Datasets_and_Benchmarks_Track.pdf
- [35] Kubernetes Documentation. 2025. *Operator pattern*. <https://kubernetes.io/docs/concepts/extend-kubernetes/operator/> [Online; accessed 31-August-2025].
- [36] Kubernetes Documentation. 2025. *The Kubernetes API*. <https://kubernetes.io/docs/concepts/overview/kubernetes-api/> [Online; accessed 31-August-2025].
- [37] Julien Lepiller, Ruzica Piskac, Martin Schäf, and Mark Santolucito. 2021. Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities. In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 105–123.
- [38] Anna Mazhar, Saad Sher Alam, William X. Zheng, Yinfang Chen, Suman Nath, and Tianyin Xu. 2025. Fidelity of Cloud Emulators: The Imitation Game of Testing Cloud-based Software. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE’25)*.
- [39] Michael Billington. 2025. *The top 100 Ansible modules*. <https://mike42.me/blog/2019-01-the-top-100-ansible-modules> [Online; accessed 31-August-2025].
- [40] Michael DeHaan. 2025. *Ansible*. <https://github.com/ansible/ansible> [Online; accessed 7-July-2025].
- [41] musl libc maintainers. 2025. *musl libc*. <https://musl.libc.org/> [Online; accessed 31-August-2025].
- [42] Ruben Opdebeeck, Bram Adams, and Coen De Roover. 2025. Analysing Software Supply Chains of Infrastructure as Code: Extraction of Ansible Plugin Dependencies. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 181–192. doi:[10.1109/SANER64311.2025.00025](https://doi.org/10.1109/SANER64311.2025.00025)
- [43] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2022. Smelly variables in Ansible infrastructure code: detection, prevalence, and lifetime. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) (*MSR ’22*). Association for Computing Machinery, New York, NY, USA, 61–72. doi:[10.1145/3524842.3527964](https://doi.org/10.1145/3524842.3527964)
- [44] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2023. Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 534–545. doi:[10.1109/MSR59073.2023.00079](https://doi.org/10.1109/MSR59073.2023.00079)
- [45] OpenRC maintainers. 2025. *The OpenRC init system*. <https://github.com/OpenRC/openrc> [Online; accessed 7-July-2025].
- [46] Paul Vixie. 2025. *Vixie Cron*. <https://salsa.debian.org/debian/cron> [Online; accessed 31-August-2025].
- [47] Yiming Qiu, Patrick Tser Jern Kon, Ryan Beckett, and Ang Chen. 2024. Unearthing Semantic Checks for Cloud Infrastructure-as-Code Programs. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) (*SOSP ’24*). Association for Computing Machinery, New York, NY, USA, 574–589. doi:[10.1145/3694715.3695974](https://doi.org/10.1145/3694715.3695974)
- [48] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2020. Gang of eight: a defect taxonomy for Infrastructure as Code scripts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (*ICSE ’20*). Association for Computing Machinery, New York, NY, USA, 752–764. doi:[10.1145/3377811.3380409](https://doi.org/10.1145/3377811.3380409)
- [49] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: security smells in Infrastructure as Code scripts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) (*ICSE*

- '19). IEEE Press, 164–175. doi:[10.1109/ICSE.2019.00033](https://doi.org/10.1109/ICSE.2019.00033)
- [50] Nuno Saavedra and João F. Ferreira. 2023. GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (, Rochester, MI, USA.) (ASE '22). Association for Computing Machinery, New York, NY, USA, Article 47, 12 pages. doi:[10.1145/3551349.3556945](https://doi.org/10.1145/3551349.3556945)
- [51] Nuno Saavedra, João Gonçalves, Miguel Henriques, João F. Ferreira, and Alexandra Mendes. 2023. Polyglot Code Smell Detection for Infrastructure as Code with GLITCH. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2042–2045. doi:[10.1109/ASE56229.2023.00162](https://doi.org/10.1109/ASE56229.2023.00162)
- [52] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: a configuration verification tool for Puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) (PLDI '16). Association for Computing Machinery, New York, NY, USA, 416–430. doi:[10.1145/2908080.2908083](https://doi.org/10.1145/2908080.2908083)
- [53] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) (MSR '16). Association for Computing Machinery, New York, NY, USA, 189–200. doi:[10.1145/2901739.2901761](https://doi.org/10.1145/2901739.2901761)
- [54] Daniel Sokolowski and Guido Salvaneschi. 2023. Towards Reliable Infrastructure as Code. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. 318–321. doi:[10.1109/ICSA-C57050.2023.00072](https://doi.org/10.1109/ICSA-C57050.2023.00072)
- [55] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. 2024. Automated Infrastructure as Code Program Testing. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1585–1599. doi:[10.1109/TSE.2024.3393070](https://doi.org/10.1109/TSE.2024.3393070)
- [56] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical fault detection in Puppet programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (ICSE '20). Association for Computing Machinery, New York, NY, USA, 26–37. doi:[10.1145/3377811.3380384](https://doi.org/10.1145/3377811.3380384)
- [57] Stephen Fromm. 2025. *ansible.builtin.user module – Manage user accounts*. https://docs.ansible.com/ansible/latest/collections/ansible/builtin/user_module.html [Online; accessed 31-August-2025].
- [58] Xudong Sun, Wenging Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing For Cluster Management Controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 143–159. <https://www.usenix.org/conference/osdi22/presentation/sun>
- [59] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. 2023. Grammar prompting for domain-specific language generation with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) (NIPS '23). Curran Associates Inc., Red Hook, NY, USA, Article 2837, 26 pages.
- [60] Wikimedia Commons. 2017. *Incident documentation/20170118-Labs*. https://wikitech.wikimedia.org/wiki/Incidents/2017-01-18_Labs [Online; accessed 7-July-2025].
- [61] Tianyi Zhang, Shidong Pan, Zejun Zhang, Zhenchang Xing, and Xiaoyu Sun. 2025. Deployability-Centric Infrastructure-as-Code Generation: An LLM-based Iterative Framework. arXiv:2506.05623 [cs.SE] <https://arxiv.org/abs/2506.05623>