State Reconciliation Defects in Infrastructure as Code

MD MAHADI HASSAN, Auburn University, USA

JOHN SALVADOR, Auburn University, USA

SHUBHRA KANTI KARMAKER SANTU, Auburn University, USA

AKOND RAHMAN, Auburn University, USA

In infrastructure as code (IaC), state reconciliation is the process of querying and comparing the infrastructure state prior to changing the infrastructure. As state reconciliation is pivotal to manage IaC-based computing infrastructure at scale, defects related to state reconciliation can create large-scale consequences. A categorization of state reconciliation defects, i.e., defects related to state reconciliation, can aid in understanding the nature of state reconciliation defects. We conduct an empirical study with 5,110 state reconciliation defects where we apply qualitative analysis to categorize state reconciliation defects. From the identified defect categories, we derive heuristics to design prompts for a large language model (LLM), which in turn are used for validation of state reconciliation.

From our empirical study, we identify 8 categories of state reconciliation defects, amongst which 3 have not been reported for previously-studied software systems. The most frequently occurring defect category is inventory, i.e., the category of defects that occur when managing infrastructure inventory. Using an LLM with heuristics-based paragraph style prompts, we identify 9 previously unknown state reconciliation defects of which 7 have been accepted as valid defects, and 4 have already been fixed. Based on our findings, we conclude the paper by providing a set of recommendations for researchers and practitioners.

CCS Concepts: • **Software and its engineering → Software defect analysis**; **Empirical software validation**.

Additional Key Words and Phrases: defect, devops, empirical study, infrastructure as code, state reconciliation

ACM Reference Format:

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 (July 2024), 24 pages. https://doi.org/10.1145/3660790

1 INTRODUCTION

Infrastructure as code (IaC) is the practice of automatically managing computing infrastructure at scale with scripts written in dedicated programming languages, such as Ansible [NIST 2023;

Authors' Contact Information: Md Mahadi Hassan, Auburn University, Auburn, Alabama, USA, mzh0167@auburn.edu; John Salvador, Auburn University, Auburn, Alabama, USA, jms0256@auburn.edu; Shubhra Kanti Karmaker Santu, Auburn University, Auburn, Alabama, USA, sks0086@auburn.edu; Akond Rahman, Auburn University, Auburn, Alabama, USA, akond@auburn.edu.

ACM 2994-970X/2024/7-ART83

https://doi.org/10.1145/3660790

Rahman et al. 2018]. The practice of IaC has gained popularity in recent years, yielding benefits for information technology (IT) organizations. For example, the use of Ansible scripts was a contributing factor for NetApp to reduce the software delivery time from days to seconds [RedHat 2022b]. As another example, the use of Ansible scripts helped the Asian Development Bank (ADB) to save hundreds of work hours while managing thousands of servers [ansible 2022].

IaC uses state reconciliation, which is unique to IaC and pivotal to manage computing infrastructure [Rahman and Parnin 2023]. In state reconciliation, a state represents the software artifacts and their corresponding configurations for the infrastructure of interest. As part of state reconciliation, an IaC orchestrator, i.e., the tool that executes IaC scripts, infers the desired infrastructure state from the script [Rahman and Parnin 2023]. Then, the orchestrator will identify the differences between the desired and existing infrastructure states, and only apply changes if there are differences between desired and existing infrastructure states [Rahman and Parnin 2023].

The importance of state reconciliation in IaC necessitates pro-active detection of defects related with state reconciliation as these defects can cause create serious consequences. Let us consider the defect shown in Listing 1 in this regard. The defect is related to state reconciliation that occurred for the Ansible orchestrator [abadger 2017]. The defect exposes sensitive information, such as passwords into log files while performing state reconciliation. According to a GitHub Advisory [GitHub Advisory Database 2022] entry, this defect is 'critical' with a severity score of 9.8 out of 10. The existence of such defects can be consequential for organizations, such as ADB who use Ansible to manage thousands of servers [RedHat 2022a]. If remained undetected and unmitigated, the defect in Listing 1 would have exposed sensitive information generated from thousands of ADB's servers.

```
- module.params.update(module.params['params'])
- # Remove the params
- module.params.pop('params', None)
...
+ if module.params['params']:
+         module.fail_json(msg="The params option
          to jenkins_plugin was removed in Ansible 2.5"
          "since it circumvents Ansible's option handling")
name = module.params['name']
state = module.params['state']
```

Listing 1. Example of a state reconciliation defect.

The existence of defects similar to Listing 1 necessitates a systematic categorization of state reconciliation defects, i.e., defects related to state reconciliation for IaC. The importance of defect categorization for software validation has already been acknowledged by the software engineering research community [Garcia et al. 2020; Humbatova et al. 2020; Rahman et al. 2020]. Categorization of state reconciliation defects, an area that remains under-explored, can be useful for researchers and practitioners in (i) understanding how state reconciliation defects occur, (ii) measuring the quality of state reconciliation implementation of IaC orchestrators, and (iii) performing validation related to state reconciliation.

Accordingly, we conduct an empirical study where we answer the following research questions:

- **RQ1:** *What categories of state reconciliation defects occur in infrastructure as code? How frequently do identified defect categories occur?*
- **RQ2:** *How can we use identified defect categories to perform validation related to state reconciliation?*

We conduct an empirical study with 5,110 state reconciliation defects mined from the open source software (OSS) Ansible orchestrator [ansible 2023]. We use multi-phase open coding [Hickey and Kipping 1996] to derive defect categories for state reconciliation. Upon derivation of defect categories, we conduct a scoping review [Arksey and O'Malley 2005] of defect-related publications to determine which of the identified defect categories have not been reported for previously studied

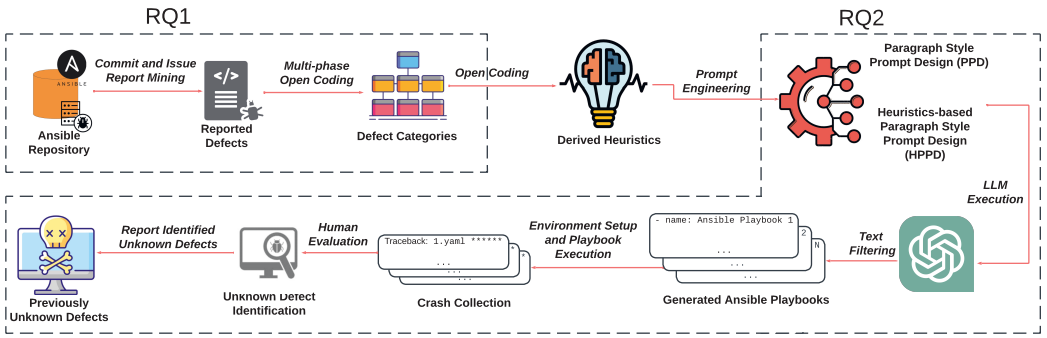

Fig. 1. An overview of our methodology.

software systems. Next, we derive heuristics for each identified defect category that are used to design prompts for GPT-3.5, a large language model (LLM) [OpenAI 2023]. These prompts are used to perform validation for state reconciliation by identifying crashes and previously unknown defects. An overview of our methodology is presented in Figure 1.

Contributions: We list our contributions as follows:

- A derived list of defects categories for state reconciliation in IaC;
- An empirical evaluation of how frequently identified defect categories occur; and
- A technique that uses identified defect categories to perform validation related to state reconciliation.

Data Availability Statement: Datasets and source code used for our paper is available online [Akond Rahman and Salvador 2023b].

2 RQ1: CATEGORIZATION OF STATE RECONCILIATION DEFECTS

We first provide the necessary background in Section 2.1. Next, in Sections 2.2 and 2.3, we, respectively, provide the methodology and answers for ***RQ1: What categories of state reconciliation defects occur in infrastructure as code? How frequently do identified state reconciliation defect categories occur?***

2.1 Background

In the case of IaC, practitioners can use configuration files called scripts, which provide necessary information for the desired state of an infrastructure. IaC orchestrators, i.e., tools that execute IaC scripts, use the state reconciliation approach during script execution. The state reconciliation process involves three steps: inventory assessment, state inquiry, and state regulation. *First*, the orchestrator will identify what the necessary inventories are, e.g., what computing clusters need to be managed along with their configurations. *Second*, the orchestrator will perform a state inquiry where the orchestrator will determine the availability and then the current state of the infrastructure. If the current state is different from the desired state as specified in the script, only then will the orchestrator perform state regulation, which involves performing necessary changes to the infrastructure as specified with the configurations specified in the script. For state regulation, the orchestrator uses auxiliaries that extend the core functionality of an orchestrator.

We use Figure 2 to demonstrate the state reconciliation approach. Figure 2a shows an example of an Ansible script that will create a file called 'simple.txt'. To execute the script, the Ansible

```yaml
---
- hosts: all
  tasks:
  - name: Creating a file with content
    copy:
      dest: "/tmp/simple.txt"
      content: |
        Line-1
        Line-2
```

a

```
[server3]
18.25.191.255
ansible_ssh_user=simple
ansible_ssh_private_key_file=<HIDDEN>/simple.pem
```

b

Fig. 2. An example to demonstrate Ansible's state reconciliation approach. Figures 2a and 2b respectively, presents that desired state expressed in YAML and the inventory infrastructure.

orchestrator will first identify the necessary inventory. The inventory for the example is presented in Figure 2b, which shows the necessary internet protocol (IP) addresses and PEM files to establish communication with an Amazon EC2 instance. Once communication is established, the orchestrator will perform state inquiry, where the orchestrator determines the current state of the infrastructure. If the current infrastructure state shows that a file called 'simple.txt' is absent then, the orchestrator will create the file as part of state regulation.

2.2 Methodology to Answer RQ1

We provide the methodology to answer RQ1 in the following subsections:

2.2.1 Detect Defects by Mining the Ansible Orchestrator Repository. We use the following steps:

Step#1 - Mine Commits and Issue Reports from the Ansible Orchestrator Repository: We download the Ansible IaC orchestrator repository [ansible 2023] on September 2022 to conduct our analysis. We use Ansible as it is one of the most popular technologies to implement IaC. Attributes of the repository is available in Table 1. From the downloaded repository, we mine commit messages from 53,195 commits and content from 31,505 issue reports. Upon mining these artifacts, we *first* apply a keyword search to identify commit messages and issue reports that are related to a defect. We use the following keywords similar to prior work [Rahman et al. 2020; Ray et al. 2014]: 'bug', 'defect', 'error', 'fault', 'fix', 'flaw', 'incorrect', 'issue', and 'mistake'. With our keyword search, we identify 22,854 commits and 2,492 issue reports that are defect-related.

Table 1. Attributes of the Downloaded Ansible Orchestrator

Attribute	Data
Commits	53,195
Contributors	5,509
Issues	31,505
Snapshot	Stable-2.14.0

Step#2 - Detect Defects by Applying Qualitative Analysis: We apply qualitative analysis to identify defects from defect-related commits and issue reports. We use qualitative analysis as only relying on keyword search can lead to false positives. We use two raters to apply qualitative analysis. One rater is the last author of the paper with 12 years of experience in software engineering. The other rater is a graduate student in the department, who is not an author of the paper. For each defect-related commit and issue report, the 2 raters individually identify if one or multiple defects appear. Both raters use the following IEEE definition [IEEE 2010] for a software defect: "*an imperfection or deficiency in the code that needs to be repaired*".

Criteria to Identify Defects - For defect identification, the raters inspected each commit message and issue report to determine if any of the following criterion is satisfied:

(1) if problematic code exists in the commit message or the issue report;

(2) if problematic code leads to an incorrect or undesired consequence that is explicitly expressed by a practitioner;

(3) the commit/issue text describes an immediate consequence of the defect; and

(4) if the problematic code was repaired.

Upon completion of the inspection process, we calculate Krippendorff's α [Krippendorff 2018] to quantify agreement, similar to prior work [Rahman et al. 2023a,b]. The Krippendorff's α is 0.62, indicating 'unacceptable' agreement [Krippendorff 2018]. The disagreements because of the second rater's misclassification when applying the above-mentioned criteria. One example issue that the second rater misclassified is "While applying these criteria, the second rater misclassified commits, which resulted in a lower agreement rate. An example misclassification is "*want to be able to use a variable for the value of ignore_errors*" [marcusphi 2013]. Both raters discussed their disagreements and identified the cause of disagreements to the perception of features or defects. Upon discussion, both raters conduct the inspection process again. After this stage, we obtain a Krippendorff's α of 1.0, indicating 'perfect' agreement [Krippendorff 2018].

2.2.2 Identify State Reconciliation Defects. As our RQ1 focuses on categorizing state reconciliation defects, we conduct another round of qualitative analysis to identify state reconciliation defects. The raters conduct closed coding [Saldaña 2015] to identify state reconciliation defects using the set of 5,898 defects from 22,854 commits and 1,263 defects from the set of 2,492 issue reports. Both raters use the following definition to determine if the content of a commit message or an issue reported can be used to label a defect as a state reconciliation defect: "*State reconciliation is defined as the approach of managing computing infrastructure by comparing the inferred state and the desired state with inventory discovery, inventory communication, state comparison, and provisioning*". For example, the defect described in the issue report titled "*Ansible command fails when we try to access it on bastion from a remote server. ssh error..Unreachable nodes*" [ansible/ansible 2023] is an example of a state reconciliation defect as it is related to Ansible's approach to communicating with a remote host that is specified as an inventory.

Criteria to Identify State Reconciliation Defects - The raters inspect if any of the following criterion is satisfied:

(1) the defect resides in orchestrator source code;

(2) the defect occurs when: (i) performing inventory discovery; (ii) performing inventory management; (iii) establishing communication with the inventory; (iv) comparing the desired and provided state; and (v) instantiating and provisioning the inventory.

We conduct closed coding in two rounds as in the first round, the Krippendorff's α is 0.41, indicating 'unacceptable' agreement [Krippendorff 2018]. The disagreements occurred because of the second rater misclassifying generic defects as state reconciliation defects. For example the commit message 'add jmainguy as author fix hash check' [1] was identified as a state reconciliation defect even though it does not follow any of the above-mentioned criterion. Prior to conducting the next round, the first rater provided the second rater necessary context on what commits and issue reports can we

[1] https://github.com/ansible/ansible/commit/b86224a7ec

classify as state reconciliation defects. In the next round, the Krippendorff's α is 0.93, which is 'acceptable' [Krippendorff 2018]. Altogether, we identified 4,410 state reconciliation defects from commits and 1,263 state reconciliation defects from issue reports. Upon elimination of duplicates, we end up with 5,110 state reconciliation defects that we use to conduct our categorization.

2.2.3 Categorization of State Reconciliation Defects. We use a qualitative analysis technique called open coding [Saldaña 2015] to derive categories for state reconciliation defects. Open coding identifies similarities in unstructured text to form categories [Saldaña 2015]. We apply open coding in two phases as multi-phase open coding [Hickey and Kipping 1996] facilitates rater reliability and achieves rater consensus. The two phases are:

Synchronized Open Coding: In this phase, the two raters categorize defects together by applying open coding with 475 defects identified from 475 issue reports and 2,080 defects identified from 2,080 defect-related commits. In this phase, the two raters discuss their rating procedures in order to achieve an acceptable level of agreement. While applying open coding, both raters inspect the commit message, commits diffs for the collected commits, along with title, description, comments, and associated code changes for the issue reports. Upon completion, the raters agree upon all but 8 defects with respect to categories. The Krippendorff's α is 0.83, indicating an 'acceptable' agreement. The disagreements are resolved by an expert in IaC with 10 years of professional experience in software engineering. The expert's categorization is final for the defects that are disagreed upon. The expert is not an author of the paper.

Independent Open Coding: In this phase, the two raters independently categorize defects by applying open coding with the remaining 2,555 defects that we do not use during synchronized open coding. Of the 2,555 defects, 475 are identified from 475 issue reports, and 2,080 defects are identified from 2,080 defect-related commits. Similar to the synchronized open coding phase, both raters inspect the commit message, commits diffs for the collected commits, along with the title, description, comments, and associated code changes for the issue reports. Upon completion, the raters agree upon all but 15 defects with respect to categories, with Krippendorff's α being 0.81, indicating 'acceptable' agreement. The disagreements are resolved by the same expert who acted as a resolver in the synchronized open coding phase. The expert's categorization is final for disagreed upon defects.

2.2.4 Comparison of State Reconciliation Defect Categories with Other Software Systems. We conduct a comparison between derived defect categories for state reconciliation in IaC and defect categories for previously studied software systems. Our assumption is that such comparison will identify defect categories unique to state reconciliation. We use two types of papers: *first*, existing taxonomies reported in the following three publications: "Orthogonal Defect Classification: A Concept for In-process Measurements" [Chillarege et al. 1992], "Bug characteristics in open source software" [Tan et al. 2014], and "Defect Categorization: Making Use of a Decade of Widely Varying Historical Data" [Seaman et al. 2008]. We select these publications as these "are seminal publications with high impact in the domain of software engineering research" [Rahman et al. 2023a]. *Second*, defect categories reported in publications accepted at the technical research tracks of the International Conference of Software Engineering (ICSE), Foundations of Software Engineering (FSE), and Journal of Systems and Software (JSS). We select ICSE, FSE, and JSS as they are well-recognized venues to publish software engineering research. We perform a scoping review [Arksey and O'Malley 2005] of papers published in the last five editions of ICSE, FSE, and JSS from 2018 to 2022. A scoping review is a reduced form of systematic literature review [Arksey and O'Malley 2005].

From our scoping review, we identify the following: "A Comprehensive Study of Autonomous Vehicle Bugs" [Garcia et al. 2020], "A Comprehensive Study on Deep Learning Bug Characteristics [Islam et al. 2019]", "A Comprehensive Study of Deep Learning Compiler Bugs" [Shen et al. 2021], "An empirical characterization of software bugs in open-source cyber-physical systems" [Zampetti et al. 2022], "An Exploratory Study of Autopilot Software Bugs in Unmanned Aerial Vehicles [Wang et al. 2021]", "An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems" [Gao et al. 2018], "An Empirical Study on Deployment Faults of Deep Learning Based Mobile Applications" [Chen et al. 2021], "An Empirical Study on Program Failures of Deep Learning Jobs" [Zhang et al. 2020], "Characterizing and Detecting Bugs in WeChat Mini-Programs" [Wang et al. 2022], "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts" [Rahman et al. 2020], "How Bad Can a Bug Get? An Empirical Analysis of Software Failures in the OpenStack Cloud Computing platform [Cotroneo et al. 2019]", "IoT Bugs and Development Challenges" [Makhshari and Mesbah 2021], "Not All Bugs Are the Same: Understanding, Characterizing, and Classifying Bug Types" [Catolino et al. 2019], "Taxonomy of Real Faults in Deep Learning Systems" [Humbatova et al. 2020], "The symptoms, causes, and repairs of bugs inside a deep learning library" [Jia et al. 2021], "Towards understanding bugs in an open source cloud management stack: An empirical study of OpenStack software bugs" [Zheng et al. 2019], "Understanding Performance Problems in Deep Learning Systems [Cao et al. 2022]", and "Using Orthogonal Defect Classification to characterize NoSQL database defects" [Agnelo et al. 2020]. In all, we use 21 publications, of which 7, 6, 5, 1, 1, and 1 are, respectively, published at ICSE, FSE, JSS, TSE, EMSE, and ESEM.

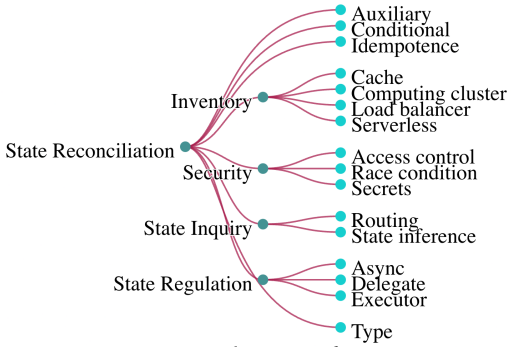

Fig. 3. State Reconciliation Defect Categories.

2.3 Answer to RQ1: Defect Categories and Their Frequency

We answer **RQ1:** *What categories of state reconciliation defects occur in infrastructure as code? How frequently do identified state reconciliation defect categories occur?* in the following subsections:

2.3.1 *Answer to RQ1: Defect Categories.*

We identify eight defect categories related to state reconciliation. An overview of the identified state reconciliation defect categories is available in Figure 3. We describe these defect categories with examples as follows:

```
value = value.rstrip()
# ...and non-printable characters
- value = filter(lambda x: x in string.printable,
  ↪ value)
+ value = ''.join(x for x in value if x in
  ↪ string.printable)
# ...tabs prevent blocks from expanding
value = value.expandtabs()
```
Listing 2. Example of an auxiliary defect, which occurred because of incorrect iteration logic for `string.printable` using `filter()`.

I **Auxiliary**: Defects that occur due to the auxiliary's inadequate handling of events. Auxiliaries extend the core functionality of an orchestrator and handle events to augment the existing state reconciliation process [Ansible 2023]. These events are typically triggered by scripts.

```
- if original_task.action in ['include',
  ↪ 'include_role']:
+ if original_task.action not in ['include',
  ↪ 'include_role']:
```
Listing 3. An example of a conditional defect where the `in` operator was incorrectly for the condition of an if block.

Example: Listing 2 shows an example of an auxiliary defect [Jim Gu 2018]. Due to this defect, a callback auxiliary fails to handle JSON objects in response to events triggered by scripts.

II **Conditional**: Defects that appear because of incorrect conditional logic.

Example: Listing 3 presents an example of a conditional logic defect, which caused incorrect display of results [bcoca 2016a].

III **Idempotence**: Defects that occur when the idempotency property is violated. Idempotency is the property that ensures that the provisioning results will be exactly the same even after multiple executions of the same IaC script [Burgess 2011; Hummer et al. 2013].

```
- child_template_ids =
↪ template.get_template_ids(link_templates)
- existing_child_templates = None
...
+ for macroitem in template_macros:
+  for key in macroitem:
+     macroitem[key] = str(macroitem[key])
```

Listing 4. An example of an idempotence defect, which occurred because of not adding code that prevents executions with diverse outcomes for the same template.

Example: Listing 4 shows an example of an idempotence defect that occurred for 'zabbix_template' [D3DeFi 2018], which is used to integrate monitoring utilities [Zabbix 2018].

IV **Inventory**: Defects that occur when managing infrastructure inventory while performing state reconciliation. We identify four sub-categories:

IV-a. *Load Balancer*: Defects that occur while performing state reconciliation for inventory related to load balancers. Load balancing is used to distribute network or application traffic across multiple servers [Cardellini et al. 1999].

Example: Listing 5 shows an example [bcoca 2016b] of an inventory-related defect when managing Amazon Elastic Load Balancers (ELBs) [Amazon 2023].

```
- elbs = elb.get_all_load_balancers()
+ elbs = []
+ marker = None
...
+ newelbs =
↪ elb.get_all_load_balancers(marker=marker)
+ elbs.extend(newelbs)
```

Listing 5. An example of an inventory defect related with load balancers that occurred because of using incorrect API methods for ELB-related inventory

.

IV-b. *Computing Clusters*: Defects that occur while performing state reconciliation for inventory related to computing and storage instances, such as Amazon Web Services (AWS) EC2 instances.

Example: Listing 6 shows an example of an inventory-related defect while installing the 'sysctl' utility [2] for OpenBSD-based computing clusters [Rick Elrod 2020].

```
def populate(self, collected_facts=None):
   hardware_facts = {}
...
- cpu_facts = self.get_processor_facts()
- memory_facts = self.get_memory_facts()
- device_facts = self.get_device_facts()
...
+ hardware_facts.update(self.get_processor_facts())
+ hardware_facts.update(self.get_memory_facts())
+ hardware_facts.update(self.get_device_facts())
```

Listing 6. An example of an inventory defect that occurred because of using the wrong API methods needed to specify hardware resources for OpenBSD-based computing clusters.

IV-c. *Cache*: Defects that occur due to incorrect caching of inventory.

Example: Listing 7 shows an example of an inventory-related defect that leads to

```
- self.module.run_command(self.yum_basecmd +
↪ ['makecache'])
+ self.module.run_command(self.yum_basecmd +
↪ ['makecache', 'fast'])
```

Listing 7. Example of a cache-related inventory defect, which occurred because of not providing `fast` for `run_command`.

[2] https://man7.org/linux/man-pages/man8/sysctl.8.html

performance issues while caching inventory [mkrizek 2021]. Because of the defect, unnecessary inventory data was being cached.

IV-d. *Serverless Inventory*: Defects that occur when managing serverless inventory [Castro et al. 2019], such as AWS Lambda [3]. AWS Lambda is a framework that allows practitioners to deploy software applications on serverless computing inventories. Unlike virtual or physical computing inventories, in the case of serverless computing inventories, practitioners are not expected to control resources, configuration management, and fault tolerance for serverless inventories [Castro et al. 2019].

```
- this_module_function = getattr(this_module,
↪  invocations[module.params['query']])
+ this_module_function =
↪  globals()[invocations[module.params['query']]]
```

Listing 8. An example of a serverless inventory defect, which occurred because of using `getattr()` instead of `globals()` that contains AWS Lambda-related data)

.

Example: Listing 8 shows an example of a defect related with serverless inventory [Jill R 2021]. Because of the defect, erroneous data was being returned to and from AWS Lambdas.

V **State Regulation**: Defects that occur during state regulation, i.e., the process when the orchestrator performs necessary changes to the desired computing infrastructure. Necessary changes that are needed are configured using the script. We identify three sub-categories:

V-a. *Async*: Defects that occur when performing state regulation in an asynchronous fashion.

```
- task_fields=self._task.dump_attrs(),
+ task_fields=async_task.dump_attrs(),
```

Listing 9. An example of a state regulation defect related to async, where `self` is used to access an async task.

Example: Listing 9 shows an example of a state regulation defect related with async [sivel 2022].

V-b. *Delegate*: Defects that occur when the orchestrator attempts to perform state regulation for an inventory that is not specified by the 'hosts' attribute.

```
- delegated_host = self._inventory.localhost
+ for h in self._inventory.get_hosts(ignore_limits=True,
↪  ignore_restrictions=True):
```

Listing 10. An example of a state regulation defect related to delegation that occurred because of not accessing delegate-related data with `self._inventory.get_hosts`.

Example: Listing 10 provides an example of a state regulation defect related with delegates [bcoca 2020].

V-c. *Executor*. Defects that occur because of incorrect code snippets, such as tasks.

Example: Listing 11 presents an example of a state regulation defect related to executor [zenbot 2016].

```
-    flag = True
-    for res in self._result.get('results', []):
-       if isinstance(res, dict):
-         flag &= res.get('skipped', False)
-       return flag
+    results = self._result['results']
+    return results and all(isinstance(res, dict) and
↪  res.get('skipped', False) for res in results)
```

Listing 11. An example of an executor-related state regulation defect because of the incorrect assumption that a task with non-dictionary loop would be skipped.

[3]https://aws.amazon.com/lambda/

VI **State Inquiry**: Defects that occur when inquiring about the state of the desired infrastructure. We identify two sub-categories:

VI-a. *Routing*: Defects that occur when routing infrastructure traffic from one destination to another. This category of defect occurs due to incorrect routing rules, incorrect binding, or incorrect specification of IP addresses and/or port numbers.

```
ipaddr = ipaddr[1:-1]
if parts[1]:
-   port_binds = [(parts[0], port) for port in
↪   parse_port_range(parts[1], self.client)]
+   port_binds = [(ipaddr, port) for port in
↪   parse_port_range(parts[1], self.client)]
```

Listing 12. An example of a state inquiry defect related to routing. The defect occurred because of using the incorrect variable (`parts[0]`) to obtain ports needed for routing.

Example: Listing 12 presents an example of a state inquiry defect related to routing [felixfontein 2019] that occurred because of not properly binding IP addresses and ports.

VI-b. *State Inference*: Inquiry defects that occur when the orchestrator fails to determine the state or incorrectly determines the state of the infrastructure. This category of defects occurs after the orchestrator becomes successful in establishing a connection with the target infrastructure.

```
- res = self.cs.expungeVirtualMachine(id=instance['id'])
+ res = self.cs.destroyVirtualMachine(id=instance['id'],
↪   expunge=True)
```

Listing 13. An example of a state inquiry defect related to inference. Because of using `expungeVirtualMachine`, Ansible incorrectly inferred instance state as 'expunged'.

Example: Listing 13 presents an example of a state inquiry defect related to inference [resmo 2016].

VII **Security**: Defects that occur due to violating the principles of confidentiality, integrity, or availability while performing state reconciliation. We identify three sub-categories:

VII-a. *Access Control*: Security-related defects that occur due to incorrect or inadequate restriction of access.

```
matched = True
if rule.get('description', None) !=
↪   r['description']:
    changed = True
- r['description'] = rule['description']
+ r['description'] = rule.get('description', None)
```

Listing 14. An example of a security defect related to access control. The defect occurred because of incorrect implementation of access control rules where an incorrect data structure (`rule`) was queried instead of using the `rule.get()` method.

Example: Listing 14 shows an example of a security defect related to access control [Zim Kalinowski 2018]. The defect occurred while setting up access control policies with security groups for Azure.

VII-b. *Secrets*: Security-related defects that occur due to inadequate management of secrets, such as user passwords and secure socket layer (SSL) certificates.

```
module = AnsibleModule(
    argument_spec = dict(
        username = dict(default=None),
-       password = dict(default=None),
+       password = dict(default=None, no_log=True),
        host = dict(default='localhost'),
```

Listing 15. An example of a security defect related to secret management. Because of not using `no_log=True`, passwords were being exposed to Ansible orchestrator logs.

Example: Listing 15 shows an example of a security defect related to secret management [mscherer 2016]. The defect resulted in the exposure of passwords.

Table 2. Comparison of Defect Categories.

Category	Previously-studied Software System
Auxiliary	Not reported for prior software systems
Conditional	Autonomous Vehicle [Garcia et al. 2020], Cyber-physical systems [Zampetti et al. 2022], Deep Learning Libraries [Islam et al. 2019], IaC Script [Rahman et al. 2020], IBM Proprietary Software [Chillarege et al. 1992], NASA Software Projects [Seaman et al. 2008], NoSQL Database [Agnelo et al. 2020], Linux Kernel [Tan et al. 2014]
Idempotence	IaC Scripts [Rahman et al. 2020]
Inventory	Not reported for prior software systems
Security	Apache Projects [Catolino et al. 2019], Autonomous Vehicle [Garcia et al. 2020], Deep learning compilers [Shen et al. 2021], Distributed Systems [Gao et al. 2018], Eclipse Projects [Catolino et al. 2019], IaC Scripts [Rahman et al. 2020], Mozilla Projects [Tan et al. 2014], Openstack [Zheng et al. 2019], WeChat [Wang et al. 2022]
State Inquiry	Apache Projects [Catolino et al. 2019], Distributed systems [Gao et al. 2018], Eclipse Projects [Catolino et al. 2019]
State Regulation	Not reported for prior software systems
Type	Cloud Computing Platform [Cotroneo et al. 2019], Deep Learning Compilers [Shen et al. 2021], Deep Learning Deployment [Chen et al. 2021; Zhang et al. 2020], Deep Learning Libraries [Islam et al. 2019; Jia et al. 2021], Deep Learning Projects [Humbatova et al. 2020]

VII-c. *Race Condition*: Security-related defects that occur due to multiple processes using the same variable or resource at a given time.

Example: Listing 16 shows an example of a security defect related to race condition [dagwieers 2019]. The defect allowed the simultaneous reading and writing of files by multiple processes.

```
- if module._diff:
-    diff['before'] = to_native(b('').join(b_lines))
# NOTE: Avoid opening the same file in this context !
+ with open_locked(dest, module.check_mode) as fd:
+    if b_lines is None:
+       b_lines = fd.readlines()
```

Listing 16. An example security defect related to race condition. The defect occurred because of not adding code that can lock a file when used by a single process.

VIII **Type**: Defects that occur due to incorrect use of types.

Example: Listing 17 shows an example of a type-related defect [bcoca 2018].

```
for b_type in ('force', 'follow', 'trim_blocks'):
   value = locals()[b_type]
-  value = ensure_type(value, 'string')
+  value = ensure_type(value, 'boolean')
```

Listing 17. An example of a type defect, where the `string` type was used instead of `boolean`

Comparison: In Table 2 we report the defect categories that have appeared for other software systems with our scoping review. As highlighted in green, we observe three categories that do not appear for previously-studied software systems: auxiliary, inventory, and state regulation.

2.3.2 Answer to RQ1: Frequency of Identified Defect Categories for State Reconciliation. We report the count of defects that belong to each category in Table 3, which is sorted alphabetically by category names. The most frequently occurring defect category is inventory. 'N/A' indicates no sub-category to exist for a category. 'Category Total' provides the total count of defects for a category with sub-categories.

> *Answer to RQ1: We derive eight defect categories for state reconciliation in IaC, of which three have not been reported in previously-studied software systems.*

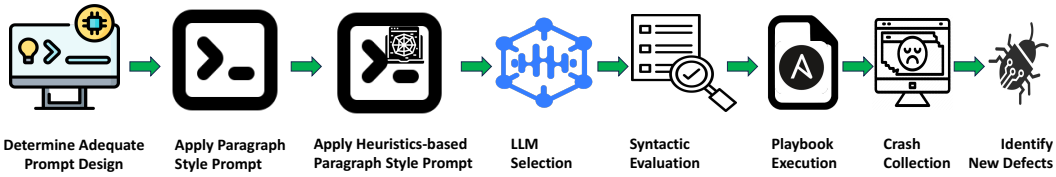

Fig. 4. An overview of our methodology to answer RQ2.

3 RQ2: STATE RECONCILIATION-RELATED VALIDATION

In this section, we answer **RQ2: How can we use identified defect categories to perform validation related to state reconciliation?** One utility of defect categorization is it provides clues on how to improve software validation efforts [Catolino et al. 2019; Chillarege et al. 1992], such as identifying new defects in the software. Accordingly, we expect our identified defect categories could aid in identifying new defects. *The goal of RQ2 is to help practitioners in identifying new state reconciliation defects by using insights from the derived defect categories from RQ1-related findings.* We accomplish our goal by generating heuristics from the derived defect categories to perform prompt engineering [Liu et al. 2023]. We observe that incorporation of these heuristics improve the prompt generation process.

We provide the methodology and results related to RQ2 respectively, in Section 3.1 and 3.2. An overview of our methodology to answer RQ2 is presented in Figure 4. In Section 3.1.1, we describe how we leverage an existing prompt design framework called 'TELeR' to generate playbooks, and how we evolved our prompt design process in order to identify new defects. We also discuss the heuristics-based prompt engineering process in Section 3.1.3.

3.1 Methodology to Answer RQ2

We use the following steps to answer RQ2:

3.1.1 Determining the Adequate Prompt Design to Automatically Generate Ansible Playbooks. RQ2 focuses on deriving a validation technique that will identify previously unknown defects related to state reconciliation. Accordingly, we generate Ansible playbooks that can identify previously unknown state reconciliation defects for the Ansible orchestrator using LLMs with prompt engineering [Liu et al. 2023]. We use LLMs for two reasons: (i) first, use of hand-crafted Ansible playbooks is manual and limiting [Cummins et al. 2018]; and (ii) second, use of existing OSS Ansible playbooks is limiting as OSS playbooks developed by practitioners solely focus on functionality, and does not account for state inference, state comparison, and script execution—all of which are pivotal to discover state reconciliation defects.

Prior research [Liu et al. 2023] has reported that LLMs' behavior widely varies based on the degree of details provided in prompts, which necessitates a systematic derivation of a prompt design technique. We apply the TELeR framework [Santu and Feng 2023] that provides guidelines on how to design prompts for LLMs. The framework includes three levels: 'no directive', 'one sentence directive', and 'paragraph-style prompt directive'. In the case of 'no directive' only data is provided with no directions [Santu and Feng 2023]. In the case of 'one sentence directive' a single and simple sentence is provided as directive [Santu and Feng 2023]. In the case of paragraph style prompt design, we use multiple sentences as prompts. To determine which level is adequate for RQ2, we randomly select 1,261 issues, and use each of the titles and bodies of issues as prompts, for each of the three levels. For each level, we quantify the syntactic correctness of generated playbooks.

The level the yields the most syntactically correct playbooks is later used as part of our paragraph style prompt design approach. From our exploration, we observe the proportion of syntactically correct playbooks is 2.2%, 34.6%, and 47.5% respectively, for 'no directive', 'one sentence directive', and 'paragraph style prompt directive'. We use paragraph style prompt directive as it provides the highest proportion of syntactic correctness.

3.1.2 Observations from Using Paragraph Style Prompt Design to Automatically Generate Ansible Playbooks. Paragraph style prompt design (PPD) is the approach of crafting detailed prompts using a description of the task along with explicitly stating the associated sub-tasks that the LLM must perform [Liu et al. 2023]. We hypothesize that the content from the title and body of each issue report can be converted into multiple sentences, each of which can be used as prompt components to construct a prompt. We extract the title and body for each issue mapped to a state reconciliation defect and convert them into multiple sentences. Each set of sentences obtained from an issue report is used as a prompt for GPT-3.5. The template that we use to generate each prompt is "*Your key task is to develop a comprehensive, self-contained Ansible playbook by taking inspiration from the given GitHub issue with title: t and the body: b.*". Here, *t* and *b*, respectively, correspond to the title and body for each of the 1,263 issues that are labeled as state reconciliation defect categories.

Table 3. Answer to RQ1: Frequency of Defect Categories.

Category	Sub-category	Count
Auxiliary	N/A	36
Conditional	N/A	184
Idempotence	N/A	179
Inventory	Cache	69
	Computing resource	1,782
	Load balancer	8
	Serverless inventory	7
	————	——
	Category Total	1,866
Security	Access control	250
	Race condition	38
	Secret management	188
	————	——
	Category Total	476
State Inquiry	State inference	562
	Routing	458
	————	——
	Category Total	1,020
State Regulation	Async	32
	Delegate	30
	Executor	597
	————	——
	Category Total	659
Type	N/A	690

In total, we use the title and body of 1,263 issue reports mapped to a state reconciliation defect to generate 1,263 prompts. Each of the 1,263 prompts generated an Ansible playbook that we used in Section 3.1.5 to perform syntactic validation. In all, we generate 1,263 playbooks of which 53.4% are syntactically correct and 51.9% are executable, as shown in Table 5.

By executing the set of 655 playbooks, we obtain 23 crashes, but none of them yielded any new defects. We further examine why the generated playbooks with PPD did not yield new defects. Our assumption is that a playbook generated with paragraph style prompt design can be syntactically correct but the content of the playbook may be inconsistent with the prompt's intention. We examine our assumption by applying closed coding [Saldaña 2015], where we select a sample of 108 playbooks that are syntactically valid to three pre-defined levels: 'irrelevant', 'somewhat relevant', and 'highly relevant'. An 'irrelevant' playbook does not follow the provided prompt. A 'somewhat relevant' playbook matches the prompt and is on topic through keyword matching. However, it is not targeted towards the goal of the prompt. A 'highly relevant' playbook matches the prompt well with the goal of the prompt provided. In all, we respectively, identify 52, 25, and 31 'irrelevant', 'somewhat relevant', and 'relevant' playbooks. Based on these evidence, we conclude that further efforts are needed to generate necessary playbooks. As such, we apply additional steps of generating heuristics from the derived taxonomy in RQ1 in order to design prompts.

Table 4. Example to Demonstrate the Heuristic Derivation for HPPD

Commit	Initial Code	Initial Heuristic	Heuristic
fix a case where we mixed text and bytes in the local connection utility	'case', 'mixed', 'text', 'bytes'	Unexpected byte strings trigger defects	Pass in byte string values instead of regular strings
fix mixing of bytes and str in replacer (caused traceback on python3)	'mixing', 'bytes', 'str', 'caused', 'traceback'	Erroneous use of byte strings cause crashes	

3.1.3 Using Heuristics-based Paragraph Style Prompt Design to Identify New State Reconciliation Defects. As part of this step, we leverage the derived categories from Section 2.3 to conduct heuristics-based paragraph style prompt design (HPPD). *First*, for each identified defect category in Section 2.3, we extract code changes that map to a category by inspecting each of the issue reports and commits that map to the defect category. *Second*, we apply open coding [Saldaña 2015] to derive heuristics by identifying similarities in the mentioned text. The open coding process is conducted by the last author. *Third*, as the open coding process is susceptible to the bias of the last author, another rater is included to perform rater verification. We use a sample 360 state reconciliation defects, which provides a confidence level of 95% for our set of 5,110 defects. From the sample of 360 defects, of which are 60 are issue reports, and the remaining 300 are commits, we extract code changes. Krippendorff's α [Krippendorff 2018] between the last author and the additional rater is 0.86, indicating 'acceptable' agreement [Krippendorff 2018]. Upon completion of this step, we will generate a set of heuristics that will be used to craft prompts, which, in turn, will be fed to the LLM.

We use Table 4 to further demonstrate the open coding process to derive heuristics. The 'Commit' column represents two commit messages: 'fix a case where we mixed text and bytes in the local connection utility' and 'fix mixing of bytes and str in replacer (caused traceback on python3)'. Both of these commit messages are labeled as type-related from our analysis of defect-related commits in Section 2.2.3. The identified codes are shown in the 'Code' column. As extracted codes express a defect-related action related to unexpected and erroneous usage of byte strings, we derive the initial heuristics 'Unexpected byte strings trigger defects' and 'Erroneous use of byte strings cause crashes', respectively, shown in row# 1 and#2. Finally, the initial heuristics are merged as as a heuristic called 'Pass in byte string values instead of regular strings'. We repeat the same process for all commits and issue reports with code changes for all eight categories.

In the case of type-related heuristics, we generate playbooks by applying an attribute-informed approach where we include an attribute name along with the heuristic. We use the following template: "*Your key task is to develop a comprehensive, self-contained Ansible playbook for the auxiliary m which performs d. This playbook aims to reveal previously unknown type-based defects, informed by a detailed understanding of the auxiliary and its unique options and attributes, including a. Your playbook should also incorporate test cases based on a specific heuristic:* h_t". Here, m is an Ansible auxiliary with a description d, where a is the set of attributes for auxiliary m. h_t refers to the type-related heuristics. We use this template as our assumption is that incorporation of attributes will aid in identifying unknown type-related defects.

For heuristics that are not related to types, we use the following template: "*Your key task is to develop a comprehensive, self-contained Ansible playbook by taking inspiration from the given GitHub issue with title:* t *and the body:* b*. This playbook aims to reveal unknown* c*-based defects in Ansible, using the issue description as a guide. Your playbook should also incorporate test cases based on the following heuristic:* h_{nt}". Here, t and b respectively, corresponds to the title and body for each of the 1,243 issues that are labeled as state reconciliation defect categories. c corresponds to each of the 8 categories, and h_{nt} corresponds to each of the identified heuristics from our open coding analysis.

For each category, c we apply the heuristic h_{nt} that is only applicable for the category c, a non-type related defect category.

3.1.4 LLM Selection. We use OpenAI's GPT that provides two versions: GPT-3.5 and GPT-4.0. We use 30 Ansible playbooks generated using PPD to determine which LLM will be adequate to generate playbooks. In the case of GPT-3.5 and GPT-4.0 we respectively, find 93.1% and 86.6% of the generated playbooks to be syntactically correct. We also observe GPT-3.5 to generate all 30 playbooks in 15 minutes, which is 2× faster than that of GPT-4.0. We select to use GPT-3.5 because (i) with respect to syntactic correctness GPT-3.5 and GPT-4.0 are comparable, and (ii) GPT-3.5 is 2× faster than that of GPT-4.0. With respect to syntactic correctness GPT-3.5 and GPT-4.0 are comparable. The main difference between GPT 4.0 and GPT 3.5 lies in GPT-4.0's ability to handle multi-modal inputs (both text and images). GPT-4.0 offers improved performance in tasks requiring comprehension of longer documents or mixed text and image data. Given our task of playbook generation, neither long documents nor image training can provide additional boost in the performance. Hence, we see similar performance with respect to syntactic correctness. With respect to execution time, GPT-3.5 is faster than that of GPT-4.0 because of model size. GPT-4.0 includes approximately 1.5 trillion parameters, whereas GPT-3.5 contains 154 billion parameters. For GPT-3.5, we use 1, 'None', and 1, as configuration values respectively, for GPT-3.5's temperature, 'logit bias', and 'top_p'.

3.1.5 Syntactic Evaluation of Generated Ansible Playbooks. LLM-generated source code files are susceptible to compilation errors [Yetistiren et al. 2022]. We use an automated approach to determine the syntactic correctness of each generated playbook. We use the command: "`ansible-playbook playbook_path -i inventory_path –private-key private_key –become-password-file become_password_file -vvv`" provided by Ansible to determine the syntactic correctness of each playbook generated with PPD and HPPD.

3.1.6 Environment Setup for Playbook Execution. Execution of the generated playbooks require setting up an environment where we can run Ansible orchestrator. *First*, we download the Ansible orchestrator (version 2.14.7) source code package [ansible 2023] with the following components: (i) *Builtin* provides default functionalities of state reconciliation; (ii) *Netcommon* performs networking-related tasks. As networking is pivotal for state reconciliation, we include this component of the Ansible orchestrator; (iii) *Utils* includes implementations of functionalities that support the core features; and (iv) *Community* includes auxiliaries developed by the Ansible community to facilitate the implementation of state reconciliation.

Second, we use a server maintained by the university to create a execution harness using Docker with four distributions: Ubuntu, Alpine, CentOS, and RedHat. The versions for Ubuntu, Alpine, CentOS, and RedHat are respectively, Ubuntu 22.04.2 LTS, Alpine Linux v3.18, CentOS Linux release 8.5.2111, and Red Hat Enterprise Linux release 8.8. We create a Docker-based network with a subnet of 10.1.1.0/24 and a gateway at 10.1.1.254. The IP assignments for the nodes are as follows: Ubuntu is at 10.1.1.1, Alpine at 10.1.1.2, CentOS at 10.1.1.3, and RedHat at 10.1.1.4. We use a network-based to replicate a typical Ansible-based environment where Ansible playbooks are executed by the orchestrator to provisioning and manage computing instances.

Table 5 provides statistics on how many of the generated Ansible playbooks are syntactically valid and are executable. We observe 53.4% and 80.0% of the generated playbooks to be syntactically valid respectively, for PPD and HPPD. We also observe 51.9% and 80.0% of the generated playbooks to execute and generate output with the execution environment described in Section 3.1.6.

3.1.7 Crash Collection and Inspection. Upon execution of the generated Ansible playbooks, we record for which playbooks crashes are generated. For each of the generated crashes, we *first*,

Table 5. Statistics of Generated Playbooks with PPD and HPPD

Approach	Playbook #	Lines of Code	Generation Time (Hours)	Syntax (%)	Executability (%)
PPD	1,263	28,605	8.7	53.4	51.9
HPPD	3,799	101,392	31.5	80.0	80.0

identify the location of the crash. *Second*, we determine if program in the Ansible orchestrator is expected to run for the provided playbook by manually inspecting the program and documentation for that program. *Third*, if the playbook of interest is expected to be executable as determined by manual analysis, we identify the crash as a symptom of an unknown defect, which is submitted as an issue report for the orchestrator.

3.1.8 Answer to RQ2. We answer RQ2 by reporting the following measurements: (i) count of crashes; and (ii) count of unknown defects generated. We repeat these measurements for all the identified defect categories from Section 2.3. We report count of crashes because crashes are indicators of unknown defects in software source code [Theisen et al. 2015]. We report count of unknown defects to evaluate if derived defect categories can be used to identify unknown defects.

3.2 Answer to RQ2

Using the HPPD approach, we identify 211 crashes. We identify 9 previously unknown defects from 9 crashes. Of the identified 9 unknown defects, 7 have been confirmed as valid defects, and 4 have already been fixed. Of the two rejected defects, one is labeled as a feature request and the other one is already fixed. A list of submitted issue reports with applicable status, i.e., 'Fixed', 'Accepted as Valid', and 'Rejected as Invalid', is available in our replication package [Akond Rahman and Salvador 2023b]. For each defect category we report the count of crashes, count of previously unknown defects, and used heuristics in Table 6 respectively, using the 'Crash #', 'Unknown Defect #', and 'Heuristic' columns. Attributes of the identified unknown defects is available in Table 7 where we tabulate identified unknown defect count based on Ansible components.

In the case of HPPD, out of 211 for 201 crashes we do not identify unknown defects. The reasons for these crashes are described below where the count of crashes is enclosed within parenthesis:

Absent Packages (134): Crashes that occur due to Ansible and/or Python libraries necessary to execute the generated playbooks. We observe two categories: (i) 45 out of 134 crashes occur due to absent Ansible packages; and (ii) 89 of the 134 crashes occur due to absent Python packages.

Playbook Semantics (28): Crashes that occur due to execution of playbooks that are syntactically correct but semantically incorrect code snippets, namely use of erroneous variable group names, division by zero computation, and incorrect Linux commands.

Inadequate Artifacts (28): Crashes that occur due to absent artifacts, namely absent files, absent databases, and artifacts with insufficient permissions.

Network (11): Crashes that occur due to network-related issues, e.g., unreachable hosts.

With PPD we only identify 7 crashes none of which yielded any previously unknown defects.

> *Answer to RQ2: Using heuristics-based paragraph style prompts we identify nine previously unknown defects in the Ansible orchestrator, of which seven have been accepted as valid defects and four have been fixed.*

Table 6. Answer to RQ2: Count of Crashes and Unknown Defects Identified with HPPD

Category	Crash #	Unknown Defect #	Heuristic
Type	188	7	i. Mixed case sensitivity for string configuration values can trigger defects ii. Pass in byte string values instead of regular strings iii. Generate configuration values that includes both bytes and strings
Inventory	7	1	i. Create symlink paths with space ii. Generate malformed Unicode strings for inventory iii. Negative cache timeout values will lead to unknown defects iv. Recreate embedded code in the issue report
State Inquiry	3	1	i. Create Base-10 problem for subnet masks ii. Create mismatching network and router names
Security	6	0	Recreate embedded code in the issue report
State Regulation	6	0	Recreate embedded code in the issue report
Idempotence	1	0	Recreate embedded code in the issue report
Auxiliary	0	0	Recreate embedded code in the issue report
Conditional	0	0	Recreate embedded code in the issue report
All	211	9	

Table 7. Answer to RQ2: Attributes of Unknown Identified Defects

Component	Defect #	Category	Status
Builtin	3	Inventory:1, Type:2	Fixed:2, Rejected: 1
Community	5	State Inquiry: 1, Type:4	Fixed:2, Accepted:2, Rejected: 1
Netcommon	1	Type:1	Accepted:1

4 DISCUSSION

We discuss the implications of our empirical study as follows:

4.1 Implications for Prompt Engineering

Results reported in Table 6 show promise with respect to HPPD being useful for identifying previously unknown state reconciliation defects. We observe that prompt engineering with PPD alone does not identify unknown state reconciliation defects. These findings lay the groundwork for further investigations in the following directions: **first**, we advocate for further research on generating better heuristics that can be used with PPD to generate Ansible playbooks that will identify unknown state reconciliation defects. The heuristic generation process can be manual or semi-automated with the possible use of information retrieval techniques, such as probabilistic retrieval [Van Rijsbergen et al. 1980]. **Second**, we advocate for further enhancements to LLM usage. Prompt design alone is limited with respect to generating Ansible playbooks that help identify unknown state reconciliation defects. One possible enhancement could be the use of fine-tuning methods or pre-trained LLMs that have been optimized for code generation, such as Code LLama [Rozière et al. 2023]. While it is a known problem that LLMs suffer from 'hallucinations' where nonsensical or incorrect text can be generated [Lin et al. 2022], LLMs could possibly be used to identify defects with playbooks that we have not included in our analysis.

4.2 Implications Related to Inventory Hardening

According to Table 3, the most frequently occurring defect category is inventory for which the Ansible orchestrator incorrectly manages inventory that needs to be provisioned. As shown in

Table 6, the heuristics-based prompt composition approach yielded a lesser amount of inventory defects compared to that of type-related defects. As such, we advocate for the generation and evaluation of novel testing techniques that may yield previously unknown inventory-related defects. Possible approaches include but are not limited to (i) grammar-based fuzzing, where the underlying grammars of different types of computing inventories can be leveraged, and (ii) cache-aware fuzzing, where the capabilities of cache management of orchestrators will be tested by performing cache-aware mutation testing.

4.3 Type Related State Reconciliation Defects

Results reported in Section 3.2 showcase that all of the seven previously unknown state reconciliation defects are type-related defects. According to Table 3, type-related defects are the third most frequent category after inventory and state inquiry. As we have only focused on crashes, we conjecture that further investigations can yield more previously unknown type-related defects for IaC orchestrators. One possible avenue of exploration could be unexpected compile time errors, where the Ansible orchestrator throws a syntax error for a syntactically valid Ansible playbook because of an underlying defect within the orchestrator. According to Chaliasos [Chaliasos et al. 2021], the frequency of unexpected compile-time errors is higher than that of crashes for type-related defects.

4.4 Generalizability Related Implications

As IaC has become the de-facto standard to automatically manage computing infrastructure, reliable state reconciliation is pivotal. If the state reconciliation approach is defective, then that can cause serious consequences. Our empirical study shows state reconciliation defects to be prevalent—as many as 5,110 defects. We have derived a taxonomy of state reconciliation defects that includes three categories of defects not reported for prior software systems. The derived defect categories helped us in identifying 9 new defects.

Genralizability of Findings Related to RQ1: While we acknowledge that our findings are obtained from Ansible, our results could be applicable for other IaC languages, such as Chef and Terraform. Similar to Ansible, Chef [chef 2009] and Terraform [hashicorp 2015] also use state reconciliation to provision and manage required infrastructure. Similar to Ansible, both Chef and Terraform orchestrators include components, such as parser and executors, using which IaC scripts written in Chef and Terraform are parsed and executed. Our methodology is technology-agnostic as it relies on commits and issue reports of IaC orchestrators. Furthermore, with respect to code quality issues, such as security weaknesses, multiple IaC languages, such as Ansible, Chef, and Terraform share commonalities [Saavedra and Ferreira 2023] that further increases the likelihood that our RQ1-related findings could be generalizable for other IaC languages, apart from Ansible.

Based on the above-mentioned observations we conjecture that our derived categories will also appear for Chef and Terraform. A replication of our methodology can provide empirical evidence to our conjecture. Anecdotally, we observe some of our identified defect categories, such as conditionals [seventieskid 2022; ttdgcp 2018], idempotency [rahulgoel1 2021], and security [mmeintker-tc 2023] to appear for other IaC orchestrators, which further substantiates our conjecture.

Genralizability of Findings Related to RQ2: Our HPPD-based methodology used for RQ2 is language-agnostic as it can be applied for other orchestrators, such as Chef and Terraform. Yet, we hypothesize differences in defect detection for non-Ansible orchestrators when our HPPD-based methodology is applied. The Ansible, Chef, and Terraform orchestrators are respectively, developed in Python, Ruby, and Go. Go is strongly-typed, whereas Python and Ruby are dynamically typed [Ray et al. 2014]. Because of these differences related to languages, our HPPD-based approach may identify

more type-related defects for Ansible and Chef, compared to that of Terraform. This statement is subject to empirical evaluation that future research can address.

5 THREATS TO VALIDITY

We discuss the limitations of our paper as follows:

Conclusion Validity: The identified defect categories for RQ1 are limited to the commit messages and issue reports that we mined from the Ansible repository. We may have missed defect categories that are reported in other IaC orchestrators. The derived heuristics are limited to the code changes that map to each identified defect category. We mitigate this limitation by generating prompts by setting the temperature of GPT-3.5 at 1.0 so that it generates a diverse set of playbooks.

Construct Validity: Our paper is susceptible to construct validity as in the case of playbook generation we execute GPT-3.5 twice. Such execution may miss generation of playbooks that are syntactically correct and potentially lead to the discovery of new defects. Also, the HPPD approach is applied differently for heuristics obtained from type and non-type defects, which can influence the defect identification process. Also, while resolving disagreements the first rater may have influenced the second rater that might in turn influence the RQ1-related findings.

External Validity: Our empirical study is limited to IaC orchestrators that are open source. Therefore, our findings may not generalize to proprietary orchestrators that are closed source. We mitigate this limitation by analyzing 5,110 defects mined from the orchestrator repository for Ansible, which is one of the most popular IaC technologies.

6 RELATED WORK

6.1 Prior Research Related with Defect Categorization

Defect categorization of software systems has gained interest amongst researchers over decades. In 1992, Chillarege et al. [Chillarege et al. 1992] proposed the Orthogonal Defect Classification (ODC) classification scheme that included eight defect categories. Since then, we observe researchers to use and extend ODC to derive defect taxonomies for other software systems. Categories proposed by Chillarege et al. [Chillarege et al. 1992] were used by Cinque et al. [Cinque et al. 2014] to categorize defects for air traffic control software. Later, in 2008, Seaman et al. [Seaman et al. 2008] extended ODC to derive 7 categories of test plan defects. Seaman et al. [Seaman et al. 2008]'s defect categorization was used and extended by Garcia et al. [Garcia et al. 2020] who identified 13 defect categories for autonomous vehicle software. Seaman et al. [Seaman et al. 2008]'s defect categorization was also used by Tan et al. [Tan et al. 2014] to categorize defects in Mozilla projects.

Use of existing defect categorization frameworks, such as ODC and Seaman et al. [Seaman et al. 2008]'s work, may be inadequate for state reconciliation, as prior research [Humbatova et al. 2020] has reported pre-defined defect categorization frameworks to be inappropriate for emerging ecosystems. As a result, researchers have also constructed bottom-up defect taxonomies for domain-specific software systems. For example, Islam et al. [Islam et al. 2019] studied 2,716 SO posts to categorize defects in deep learning libraries, such as Keras and Tensorflow. Humbatova et al. [Humbatova et al. 2020] mined GitHub issues and SO posts to derive a fault taxonomy for software projects that use deep learning. Makhshari and Mesbah [Makhshari and Mesbah 2021] mined 5,565 issue reports to derive a defect taxonomy for the internet of things (IoT) software projects. Chen et al. [Chen et al. 2021] used Stack Overflow posts to derive a taxonomy of defects for deep learning-based deployment in mobile apps. Cotreno et al. [Cotroneo et al. 2013], Gao et al. [Gao et al. 2018], Shen et al. [Shen et al. 2021], and Rahman et al. [Rahman et al. 2020] constructed

defect taxonomies in a bottom-up fashion with qualitative analysis respectively, for OpenStack, distributed systems, deep learning compilers, and IaC scripts.

6.2 Prior Research Related with Quality Aspects of IaC

In recent years, quality assurance for IaC has garnered a lot of interest among researchers where majority of techniques related to quality assurance use coding pattern-based approaches. Researchers [Sharma et al. 2016] have studied code properties that cause maintainability problems. Researchers have also derived a catalog of code properties in IaC scripts [Dalla Palma et al. 2022] that correlate with defects. Defect-related research was also conducted by Shambaugh et al. [Shambaugh et al. 2016] to identify non-determinism defects, and by Sotiropoulos et al. [Sotiropoulos et al. 2020] to identify dependency-related defects in Puppet scripts. Researchers have also investigated security weaknesses for IaC scripts, such as Ansible scripts [Opdebeeck et al. 2023; Saavedra and Ferreira 2023], Chef scripts [Saavedra and Ferreira 2023], and Puppet scripts [Rahman and Parnin 2023; Reis et al. 2023; Saavedra and Ferreira 2023].

In short, we observe a lack of research that has systematically characterized state reconciliation defects in IaC. We address this gap by (i) categorizing state reconciliation defects, and (ii) using the categorization to automatically identify unknown state reconciliation defects in IaC orchestrators.

7 CONCLUSION

Similar to script development and execution, reliable implementation of state reconciliation is pivotal to reliable IaC-based infrastructure management. However, state reconciliation defects hinder the reliability of IaC-based infrastructure provisioning and management. A characterization of state reconciliation defects can aid in gaining an understanding of state reconciliation defects and also yield insights on how to perform validation for state reconciliation by identifying previously unknown state reconciliation defects. We have conducted an empirical study with 5,110 state reconciliation defects mined from the OSS repository of the Ansible orchestrator. We have derived a taxonomy for state reconciliation defects where we identify 8 defect categories, of which 3 have not been reported in prior research related to software defect taxonomies. By applying a heuristics-based prompt design approach, we identify 9 previously unknown defects, of which 7 have been accepted as valid defects, and 4 have already been fixed. Our paper lays the groundwork for future research that can identify previously unknown state reconciliation defects.

DATA AVAILABILITY STATEMENT

The artifact for the paper is publicly-available online [Akond Rahman and Salvador 2023a].

ACKNOWLEDGMENTS

We thank the PASER group at Auburn University for their valuable feedback. This research was partially funded by the U.S. National Science Foundation (NSF) Award # 2247141, Award # 2312321, and the U.S. National Security Agency (NSA) Award # H98230-21-1-0175. This work has benefited from Dagstuhl Seminar 23082 'Resilient Software Configuration and Infrastructure Code Analysis.'

REFERENCES

abadger. 2017. jenkins_plugin "params" argument is insecure. https://github.com/ansible/ansible/issues/30874. [Online; accessed 30-March-2024].

João Agnelo, Nuno Laranjeiro, and Jorge Bernardino. 2020. Using orthogonal defect classification to characterize nosql database defects. *Journal of Systems and Software* 159 (2020), 110451.

Md Mahdi Hassan Akond Rahman and John Salvador. 2023a. Artifact for Paper. 10.6084/m9.figshare.24129996.v1. [Online; accessed 19-April-2024].

Md Mahdi Hassan Akond Rahman and John Salvador. 2023b. Verifiability Package for Paper. 10.6084/m9.figshare.24129996. [Online; accessed 15-April-2024].

Amazon. 2023. Elastic Load Balancing. https://aws.amazon.com/elasticloadbalancing/. [Online; accessed 24-March-2023].

ansible. 2022. ADB uses Red Hat Ansible Automation Platform to boost infrastructure management. https://www.redhat.com/en/resources/asian-development-bank-case-study. [Online; accessed 25-Sep-2023].

Ansible. 2023. Ansible Documentation. https://docs.ansible.com/. [Online; accessed 19-December-2022].

ansible. 2023. ansible/ansible. https://github.com/ansible/ansible. [Online; accessed 25-Sep-2023].

ansible/ansible. 2023. Ansible command fails when we try to access it on bastion from remote server. ssh error..Unreachable nodes. https://github.com/ansible/ansible/issues/45898. [Online; accessed 28-March-2023].

Hilary Arksey and Lisa O'Malley. 2005. Scoping studies: towards a methodological framework. *International Journal of Social Research Methodology* 8, 1 (2005), 19–32. https://doi.org/10.1080/1364557032000119616 arXiv:https://doi.org/10.1080/1364557032000119616

bcoca. 2016a. fixed bad condition hiding results. https://github.com/ansible/ansible/commit/65c373c. [Online; accessed 20-March-2023].

bcoca. 2016b. loop to get all load balancers, boto limited to 400 at a time fixes. https://github.com/ansible/ansible/commit/90d084d. [Online; accessed 23-March-2023].

bcoca. 2018. Ensure string types (#42362) * actually enforce string types. https://github.com/ansible/ansible/commit/4a7940c. [Online; accessed 10-March-2023].

bcoca. 2020. fix delegation vars usage (debug still shows inventory_hostname). https://github.com/ansible/ansible/commit/2165f9a. [Online; accessed 26-March-2023].

Mark Burgess. 2011. Testable System Administration. *Commun. ACM* 54, 3 (March 2011), 44–49. https://doi.org/10.1145/1897852.1897868

Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding Performance Problems in Deep Learning Systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 357–369. https://doi.org/10.1145/3540250.3549123

Valeria Cardellini, Michele Colajanni, and Philip S Yu. 1999. Dynamic load balancing on web-server systems. *IEEE Internet computing* 3, 3 (1999), 28–39.

Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Commun. ACM* 62, 12 (nov 2019), 44–54. https://doi.org/10.1145/3368454

Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not All Bugs Are the Same: Understanding, Characterizing, and Classifying Bug Types. *J. Syst. Softw.* 152, C (jun 2019), 165–181. https://doi.org/10.1016/j.jss.2019.03.002

Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-typed programs can go wrong: A study of typing-related bugs in jvm compilers. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.

chef. 2009. chef/chef: Chef. https://github.com/chef/chef. [Online; accessed 17-Feb-2024].

Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An Empirical Study on Deployment Faults of Deep Learning Based Mobile Applications. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 674–685. https://doi.org/10.1109/ICSE43902.2021.00068

Ram Chillarege, Inderpal Bhandari, Jarir Chaar, Michael Halliday, Diane Moebus, Bonnie Ray, and Man-Yuen Wong. 1992. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering* 18, 11 (Nov 1992), 943–956. https://doi.org/10.1109/32.177364

Marcelo Cinque, Dominico Cotroneo, Raffaele D. Corte, and Antonio Pecchia. 2014. Assessing Direct Monitoring Techniques to Analyze Failures of Critical Industrial Systems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 212–222. https://doi.org/10.1109/ISSRE.2014.30

Domenico Cotroneo, Luigi De Simone, Pietro Liguori, Roberto Natella, and Nematollah Bidokhti. 2019. How Bad Can a Bug Get? An Empirical Analysis of Software Failures in the OpenStack Cloud Computing Platform. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 200–211. https://doi.org/10.1145/3338906.3338916

Domenico Cotroneo, Roberto Pietrantuono, and Stefano Russo. 2013. Testing techniques selection based on ODC fault types and software metrics. *Journal of Systems and Software* 86, 6 (2013), 1613–1637.

Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 95–105. https://doi.org/10.1145/3213846.3213848

D3DeFi. 2018. zabbix_template: fixed idempotency issues. https://github.com/ansible/ansible/commit/a9aa105. [Online; accessed 21-March-2023].

dagwieers. 2019. Use locking for concurrent file access. https://github.com/ansible/ansible/commit/e152b277cfc055a3b7bfdaa41db024168ca7a2a. [Online; accessed 31-March-2023].

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. https://doi.org/10.1109/TSE.2021.3051492

felixfontein. 2019. docker_container: fix port bindings with IPv6 addresses. https://github.com/ansible/ansible/commit/a757310. [Online; accessed 28-March-2023].

Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 539–550. https://doi.org/10.1145/3236024.3236030

Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, Chen, and Qi Alfred. 2020. A Comprehensive Study of Autonomous Vehicle Bugs. 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, 385–396. https://doi.org/10.1145/3377811.3380397

GitHub Advisory Database. 2022. Ansible Insertion of Sensitive Information into Log File vulnerability. https://github.com/advisories/GHSA-588w-w6mv-3cw5. [Online; accessed 31-March-2024].

hashicorp. 2015. hasicorp/terraform - Terraform. https://github.com/hashicorp/terraform/. [Online; accessed 16-Feb-2024].

Gary Hickey and Cheryl Kipping. 1996. A multi-stage approach to the coding of data from open-ended questions. *Nurse researcher* 4, 1 (1996), 81–91.

Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. 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, 1110–1121. https://doi.org/10.1145/3377811.3380395

Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing idempotence for infrastructure as code. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 368–388.

IEEE. 2010. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)* (Jan 2010), 1–23. https://doi.org/10.1109/IEEESTD.2010.5399061

Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935.

Jill R. 2021. Fix to return data when using lambda_info module. https://github.com/ansible/ansible/commit/6fa070e82. [Online; accessed 25-March-2023].

Jim Gu. 2018. yaml callback fails on python3. https://github.com/ansible/ansible/commit/5839f07. [Online; accessed 19-March-2023].

Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications.

Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. TruthfulQA: Measuring How Models Mimic Human Falsehoods. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 3214–3252. https://doi.org/10.18653/v1/2022.acl-long.229

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (jan 2023), 35 pages. https://doi.org/10.1145/3560815

Amir Makhshari and Ali Mesbah. 2021. IoT Bugs and Development Challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 460–472. https://doi.org/10.1109/ICSE43902.2021.00051

marcusphi. 2013. want to be able to use a variable for the value of ignore_errors. https://github.com/ansible/ansible/issues/4892. [Online; accessed 18-Feb-2024].

mkrizek. 2021. yum: avoid storing unnecessary cache data. https://github.com/ansible/ansible/commit/461f30c. [Online; accessed 25-March-2023].

mmeintker-tc. 2023. Terraform ignores skip_credentials_validation flag for s3 backend with custom endpoint. https://github.com/hashicorp/terraform/issues/33983. [Online; accessed 15-Feb-2024].

mscherer. 2016. Do not leak mail password by error. https://github.com/ansible/ansible/commit/b8706a1. [Online; accessed 31-March-2023].

NIST. 2023. infrastructure as code. https://csrc.nist.gov/glossary/term/infrastructure_as_code. [Online; accessed 25-Sep-2023].

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. https://doi.org/10.1109/MSR59073.2023.00079

OpenAI. 2023. GPT-4 Technical Report. http://arxiv.org/abs/2303.08774 arXiv:2303.08774 [cs].

Akond Rahman, Dibyendu Brinto Bose, Raunak Shakya, and Rahul Pandita. 2023a. Come for Syntax, Stay for Speed, Understand Defects: An Empirical Study of Defects in Julia Programs. *Empirical Software Engineering* 28, 93 (2023), 33.

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. https://doi.org/10.1145/3377811.3380409 pre-print: https://akondrahman.github.io/papers/icse20_acid.pdf.

Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. 2018. A systematic mapping study of infrastructure as code research. *Information and Software Technology* (2018). https://doi.org/10.1016/j.infsof.2018.12.004

Akond Rahman and Chris Parnin. 2023. Detecting and Characterizing Propagation of Security Weaknesses in Puppet-Based Infrastructure Management. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3536–3553. https://doi.org/10.1109/TSE.2023.3265962

Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. 2023b. Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 99 (May 2023), 36 pages. https://doi.org/10.1145/3579639

rahulgoel1. 2021. Yum package idempotency fixes. https://github.com/chef/chef/issues/12382. [Online; accessed 12-Feb-2024].

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 155–165. https://doi.org/10.1145/2635868.2635922

RedHat. 2022a. Customer Case Study - NEC. https://www.ansible.com/hubfs/pdf/Ansible-Case-Study-NEC.pdf. [Online; accessed 12-Sep-2023].

RedHat. 2022b. Customer Case Study - NetApp. https://www.ansible.com/hubfs/2018_Content/RH-netapp-case-study.pdf. [Online; accessed 02-Sep-2023].

Sofia Reis, Rui Abreu, Marcelo d'Amorim, and Daniel Fortunato. 2023. Leveraging Practitioners' Feedback to Improve a Security Linter. 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 66, 12 pages. https://doi.org/10.1145/3551349.3560419

resmo. 2016. cloudstack: fix state=expunged in cs_instance. https://github.com/ansible/ansible/commit/4020ebaecff. [Online; accessed 29-March-2023].

Rick Elrod. 2020. sysctl/openbsd fact fixes. https://github.com/ansible/ansible/commit/7094849. [Online; accessed 24-March-2023].

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. (2023).

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. https://doi.org/10.1145/3551349.3556945

Johnny Saldaña. 2015. *The coding manual for qualitative researchers.* Sage.

Shubhra Kanti Karmaker Santu and Dongji Feng. 2023. TELeR: A General Taxonomy of LLM Prompts for Benchmarking Complex Tasks. arXiv:2305.11430 [cs.AI]

Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. 2008. Defect Categorization: Making Use of a Decade of Widely Varying Historical Data. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Kaiserslautern, Germany) *(ESEM '08)*. Association for Computing Machinery, New York, NY, USA, 149–157. https://doi.org/10.1145/1414004.1414030

seventieskid. 2022. Invalid index - Output from a conditional resource contained in a module. https://github.com/hashicorp/terraform/issues/32044. [Online; accessed 09-Feb-2024].

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. https://doi.org/10.1145/2908080.2908083

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)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/2901739.2901761

Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. https://doi.org/10.1145/3468264.3468591

sivel. 2022. Resolve perf issue with async callback events. https://github.com/ansible/ansible/commit/96ce480. [Online; accessed 25-March-2023].

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. https://doi.org/10.1145/3377811.3380384

Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19 (2014), 1665–1705.

Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. 2015. Approximating Attack Surfaces with Stack Traces. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 199–208. https://doi.org/10.1109/ICSE.2015.148

ttdgcp. 2018. Chef14: broken powershell version check. https://github.com/chef/chef/issues/7166. [Online; accessed 11-Feb-2024].

Cornelis J Van Rijsbergen, Stephen Edward Robertson, and Martin F Porter. 1980. *New models in probabilistic information retrieval*. Vol. 5587. British Library Research and Development Department London.

Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. 2021. An Exploratory Study of Autopilot Software Bugs in Unmanned Aerial Vehicles. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 20–31. https://doi.org/10.1145/3468264.3468559

Tao Wang, Qingxin Xu, Xiaoning Chang, Wensheng Dou, Jiaxin Zhu, Jinhui Xie, Yuetang Deng, Jianbo Yang, Jiaheng Yang, Jun Wei, and Tao Huang. 2022. Characterizing and Detecting Bugs in WeChat Mini-Programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 363–375. https://doi.org/10.1145/3510003.3510114

Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the Quality of GitHub Copilot's Code Generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering* (Singapore, Singapore) *(PROMISE 2022)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/3558489.3559072

Zabbix. 2018. Monitoring and Integration Solutions. https://www.zabbix.com/integrations?cat=monitoring_systems. [Online; accessed 22-March-2023].

Fiorella Zampetti, Ritu Kapur, Massimiliano Di Penta, and Sebastiano Panichella. 2022. An empirical characterization of software bugs in open-source cyber-physical systems. *Journal of Systems and Software* 192 (2022), 111425.

zenbot. 2016. Don't assume a task with non-dict loop results has been skipped.). https://github.com/ansible/ansible/commit/85868e07a9a4641c845ad1be3d036e716ff89bad. [Online; accessed 27-March-2023].

Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An Empirical Study on Program Failures of Deep Learning Jobs. 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, 1159–1170. https://doi.org/10.1145/3377811.3380362

Wei Zheng, Chen Feng, Tingting Yu, Xibing Yang, and Xiaoxue Wu. 2019. Towards understanding bugs in an open source cloud management stack: An empirical study of OpenStack software bugs. *Journal of Systems and Software* 151 (2019), 210–223.

Zim Kalinowski. 2018. fix for security group description crash. https://github.com/ansible/ansible/commit/5d2c23e2a3ec9ed81a4cbb8bd6bf28785fbadf4d. [Online; accessed 30-March-2023].

Received 2023-09-28; accepted 2024-04-16