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.


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

ACM Reference Format:


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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2994-970X/2024/7-ART83
https://doi.org/10.1145/3660790

Proc. ACM Softw. Eng., Vol. 1, No. FSE, Article 83. Publication date: July 2024.
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 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.

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

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.

The 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 are 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 Repository

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commits</td>
<td>53,195</td>
</tr>
<tr>
<td>Contributors</td>
<td>5,509</td>
</tr>
<tr>
<td>Issues</td>
<td>31,505</td>
</tr>
<tr>
<td>Snapshot</td>
<td>Stable-2.14.0</td>
</tr>
</tbody>
</table>

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”.

Proc. ACM Softw. Eng., Vol. 1, No. FSE, Article 83. Publication date: July 2024.
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 $\alpha$ [Krippendorff 2018] to quantify agreement, similar to prior work [Rahman et al. 2023a,b]. The Krippendorff’s $\alpha$ 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 $\alpha$ 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 $\alpha$ 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’ [https://github.com/ansible/ansible/commit/b86224a7ec] 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
classify as state reconciliation defects. In the next round, the Krippendorff’s $\alpha$ 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 $\alpha$ 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 $\alpha$ 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].

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:

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.
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].

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

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].

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 for OpenBSD-based computing clusters [Rick Elrod 2020].

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 incorrect caching of inventory.


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.

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

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.

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

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.

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.

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].

3https://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.

Example: Listing 12 presents an example of a state inquiry defect related to routing [felixfontein 2019] that occurred because of using the incorrect variable (parts[0]) to obtain ports needed for routing.

```python
1. ipaddr = ipaddr[1:-1]
2. if parts[1]:
3. - port_binds = [(parts[0], port) for port in parse_port_range(parts[1], self.client)]
4. + 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.

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.

Example: Listing 13 presents an example of a state inquiry defect related to inference [resmo 2016] that occurred because of not properly binding IP addresses and ports.

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


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.

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.

```python
1. matched = True
2. if rule.get('description', None) !=
3.   r['description']:
4.   changed = True
5. - r['description'] = rule['description']
6. + 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.

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

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

```python
1. module = AnsibleModule(
2.   argument_spec = dict(
3.     username = dict(default=None),
4.     password = dict(default=None, no_log=True),
5.     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.
Table 2. Comparison of Defect Categories.

<table>
<thead>
<tr>
<th>Category</th>
<th>Previously-studied Software System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auxiliary</td>
<td>Not reported for prior software systems</td>
</tr>
<tr>
<td>Idempotence</td>
<td>IaC Scripts [Rahman et al. 2020]</td>
</tr>
<tr>
<td>Inventory</td>
<td>Not reported for prior software systems</td>
</tr>
<tr>
<td>State Inquiry</td>
<td>Apache Projects [Catolino et al. 2019], Distributed systems [Gao et al. 2018], Eclipse Projects [Catolino et al. 2019]</td>
</tr>
<tr>
<td>State Regulation</td>
<td>Not reported for prior software systems</td>
</tr>
</tbody>
</table>

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.

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

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

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.

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.
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.

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 does not follow the provided prompt. A ‘somewhat 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.
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 $\alpha$ [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 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 2x 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 2x 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-3.5 and GPT-4.0 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,
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

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

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

<table>
<thead>
<tr>
<th>Component</th>
<th>Defect #</th>
<th>Category</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Builtin</td>
<td>3</td>
<td>Inventory:1, Type:2</td>
<td>Fixed:2, Rejected: 1</td>
</tr>
<tr>
<td>Community</td>
<td>5</td>
<td>State Inquiry: 1, Type:4</td>
<td>Fixed:2, Accepted:2, Rejected: 1</td>
</tr>
<tr>
<td>Netcommon</td>
<td>1</td>
<td>Type:1</td>
<td>Accepted:1</td>
</tr>
</tbody>
</table>

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.

Generalizability of RQ1-related Findings: 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; ttgcp 2018], idempotency [rahulgoel1 2021], and security [mmeintker-tc 2023] to appear for other IaC orchestrators, which further substantiates our conjecture.

Generalizability of RQ2-related Findings: 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. Cotroneo 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


Proc. ACM Softw. Eng., Vol. 1, No. FSE, Article 83. Publication date: July 2024.


zenbot. 2016. Don’t assume a task with non-dict loop results has been skipped.). https://github.com/ansible/ansible/commit/85868e07a9a46e1c845ad1be03d036c716ff89bad. [Online; accessed 27-March-2023].


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