A study of bugs found in the Ansible configuration management system

Author: Matas Rastenis
Supervisor: Thodoris Sotiropoulos
Responsible Professor: Diomidis Spinellis
EEMCS, Delft University of Technology, The Netherlands

Sunday 19th June, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Research that focuses on examining software bugs is critical when developing tools for preventing and for fixing software issues. Previous work in this area has explored other types of systems, such as bugs of compilers and security issues stemming from open source systems hosted on public repositories. This paper explores the bugs within the Ansible software provisioning and configuration management system. The main question this paper seeks to answer is "What common patterns can be extracted from the bugs found and what are the root causes, symptoms, triggers, system-dependence factors, fixes, and the impact of the most frequent types of bugs in the Ansible configuration management system". This study defines a data pipeline and custom tools to extract and analyze 100 Ansible bugs. Common classifications are determined, and the bugs are manually classified, revealing common patters within the bugs. Insights are drawn from the aggregated data, and recommendations are made for addressing bug-prone areas of execution and connectivity components, and expanding the test suite with input fuzzing and a genetic algorithms test solution, in order to improve the overall code quality of the Ansible code base.

1 Introduction

In the software world, quality assurance is extremely important in order to produce reliable, deterministic and highly productive applications. Quality assurance consists of a multitude of factors, chief among them being code quality and by extension the absence of bugs and other issues. Research that focuses on examining software bugs is instrumental in understanding the ways in which systems fall prey to such problems, and to what extent the damage can be controlled after discovery. In order to better prepare for bugs and to build measures for preventing them during software development, we must first examine in depth how these bugs manifest [1]. In addition, we can use this knowledge to go further and develop sophisticated models for avoiding the development paths that generally lead to buggy code. Moreover, we can also use this research for developing methods to effectively find bugs after they have manifested, since finding and fixing the issues ultimately contributes to improving the overall quality and maintainability of the software at hand [2]. The foundational aim here is to help improve the quality of software in general.

Software quality is even more important in configuration management systems, since they are mission critical and control fleets of other physical compute hosts, so even a minor failure can prove to be catastrophic. One of such systems, Ansible, is examined in this paper. Ansible is a software provisioning and configuration management tool that is open source and provides means for developers to describe their infrastructure as code [3].

Upon examining existing literature, a limited amount of research regarding specifically Ansible bugs is present. There exists an in-depth examination of the bugs found within the user-provided Ansible scripts, found in open source repositories[4], however user-created bugs that are not within Ansible core or in any Ansible module are not the focus of this paper, since the main goal is to assure quality of the underlying system. There are also some recent papers on possible security issues stemming from users employing sub-optimal practices in their code bases [5], however that does not indicate any unintended behaviour occurred within the software itself, which is the focus of this study.

For this study, the research question is "What common patterns can be extracted from the bugs found and what are the root causes, symptoms, triggers, system-dependence factors, fixes, and the impact of the most frequent types of bugs in the Ansible configuration management system". This is a reasonable hypothesis to test, because it covers all of the main data points of the process of discovering how bugs manifest within Ansible. All of the constituent parts of the question, the common patterns from looking at the types of bugs in combination with the root causes - what actually caused the bugs, the symptoms - how did the bugs originally manifest, the root causes - what were the causes of the bugs, the triggers - what factors made the bugs appear in the first place, the fixes - how were the bugs addressed, and the impact - what the implications of the bugs were, give us solid ground on which to base this study on. The result after this study will be a detailed overview of all of the aforementioned factors, combined into a paper that can be of use for examining the commonalities of the bugs within this particular system [6], and can be utilized as a foundation for further research in the aforementioned field of linking changes to bugs [2], and to further examine the causes of bugs in software in general [1].

The main research question can be broken down into a set of sub-questions:

1. **RQ1. What are the common patterns of Ansible bugs?**
    - Are there a set of bug classes that can be used to categorize all bugs?
    - How can these findings be used to improve the code quality inside the Ansible code base?
2. **RQ2. What are the main symptoms of common Ansible bugs?**
    - How does the bug manifest?

3. **RQ3. What are the main root causes of common Ansible bugs?**
   - Which part of the system failed?
4. **RQ4. What is the impact of common Ansible bugs?**
   - What is the severity of the bug?
   - What are the consequences of the bug?
5. **RQ5. What are the fixes of common Ansible bugs?**
6. **RQ6. Are Ansible bugs system dependent?**
7. **RQ7. What are the triggers of common Ansible bugs?**
   - What action created the bug?

The main contributions of this study are a recommendation of various testing techniques such as genetic algorithm testing and specific vulnerable input fuzzing to improve the code quality of the Ansible code base [7, 8]. In addition, the most bug-prone execution and connectivity components of Ansible are outlined as being the best investments in terms of targeted code quality improvements, in terms of expected returns.

The rest of the paper will describe the methodology, the analysis itself, the results and the insights drawn from the study.

2 Methodology

This section contains the methodology used in this study in the form of the overall strategy, and the concrete implementation for analysis.

2.1 Strategy

The strategy for completing the task at hand can be split into multiple distinct sections.

Firstly, a source for Ansible bugs will be identified. Custom tooling will be made to fetch data from that source. The result will be raw data in some format.

The second step is to construct a pipeline to convert the raw data into a format that can be parsed by further analysis tools. In addition, all unnecessary data will be pruned to reduce the disk footprint for the raw and processed data. A suitable unifying format will be picked for the inter-step data storage. In addition, a programming language will be selected for writing tooling for data manipulation and aggregation.

The third step is to ensure all data are present and are mapped correctly in-between. The first sub-step is to ensure all of the required data has been pulled, and that enough of the data has been pulled. Then, the second sub-step is to map the data correctly, to ensure analysis can be done. This would constitute mapping bug reports to fix proposals of the bugs, and mapping them to the related pull requests that implement the fixes. This step also has to contain the filtering of unfit bugs.

The fourth step is to serialize the data into a database. This step ensures only the sufficient minimum amount of data is kept, and it allows for centralized and standardized data management for bugs of all kinds. For this study, other configuration management systems will have to arrive at a convergent database schema, to ensure all bugs and fixes can be co-located within the same shared database. This also allows for more extensive options for querying the data.

The fifth step is the analysis of the issues. The issues can now be queried and analyzed. Common patterns can be identified, and the resulting findings can be extracted and presented in the study. This will open up the possibilities of identifying problematic fields and areas for future improvement for the Ansible configuration management system.

The sixth step is to represent the findings in a clear format, and to provide visuals.

This concludes the series of steps that constitute the methodology.

2.2 Bug Collection

The implementation of the strategy described previously is as follows.

The implementation follows the strategy described earlier in this section. This sub-section will contain the exact requisite implementation for performing the analysis.

The officially recognized source for bug tracking, Ansible GitHub Issues, was used to obtain all of the relevant data [9]. Custom tooling was created to pull, filter and serialize raw data. Then, sorting by type produced the initial grouping. The next step is the analysis the bugs in accordance with the research question. The final steps were to aggregate the data and to draw conclusions about common patterns in Ansible bugs.

First and foremost, a tool was identified for fetching of raw data from Github Issues. After evaluating available options, the Perceval project [10] was deemed to be the most apt solution. The reason for picking it is that it provides a way to fetch data from Github in a standardized way and with the required filtering options included. This is a preferred approach to using the Github API as is and building superfluous custom tooling for raw data fetching. Perceval was used in bash scripts to create the final utilities for pulling raw data. The scripts pullRawIssues.sh and pullRawPRs.sh are utilized to download raw data from Github, utilizing Perceval. Before attempting to execute these scripts, one must set up the environmental variables, namely the Github API key. The full list can be found in the .env.example file. These need to be set up prior to beginning data fetching.

The second step is to pre-process the data into a parsable format, and to prune unneeded data. To facilitate this, tooling was created that parses the raw data pulled with the scripts from step 1. The resulting output is of the JSON array format, and contains only the necessary fields. The tooling in this step and future steps is implemented in Python. Python was picked for it's versatility, readability, and low infrastructure overhead, since the scripts can be immediately run by anyone that has Python 3 installed. Next comes the filtering stage, to clean up various types of issues that are not suitable for analysis, and to produce a preliminary summary from the data. This is done in the extractSummary.py script. A summary of bug issue names, labels, PR links, as well as the total counts is displayed.

The third step is to serialize the data into a MySQL database with a shared schema. The shared schema was decided upon together with the other research project teammates to contain all necessary data for a bug to be evaluated. The serializeIntoDB.py script takes the filtered and pruned output and inserts it into a database. This is done to collect all of the data in s standardized way, as well as to have more extensive options for querying the data.

The fourth step is the sampling of issues, implemented in the script sample.py. Here, the goal is to extract a set amount of issues at random, for the purposes of analysis. They are picked at random from the output of the fourth step, and displayed to the user in a list form.

The fifth step is to begin analyzing the bugs by answering each of the research questions for the bugs sampled in the fifth step. While doing this, a set amount of categorization values will begin to emerge for each of the research questions. It is important to create concise, meaningful and enumerable answers for each question. Then, each bug can be analyzed and categorized with respect to the answers of each research question. This produces the

The sixth and final step is to analyze all remaining issues using the categorization framework from step six, and to produce visualizations of the resulting data. The output is the final categorization of all sampled bugs, as well as a detailed overview of the data in the form of visualizations. This will allow for discovery of common patterns throughout the sampled bugs and for the drawing of insights on how to improve the code base going forward.

3 Bug Analysis

This section contains an outline of what the main ideas of this study are, and how they will be able to be utilized to deliver improvements for real live applications, and how they will improve on already existent studies and data.

The main idea is that software quality can be improved by analyzing the overarching issues in a system, and by extracting common patterns from them in order to develop ways to prevent such issues from occurring in the future, thus improving the stability and overall quality of the software. In order to be able to obtain these insights, aspects of interest for issues must first be outlined, which was done in the research question. The research question along with the sub-questions was laid out in the introduction section. The goal now is to answer the research question.

This study scopes in on the configuration management system sphere, as it is an area where system failures have great consequences, and thus software quality must be taken very seriously. Specifically, the Ansible configuration management system. There are no existing studies that contain analysis of the bugs found in the Ansible core code, so to fill that information gap, it is the aim of this study. This advances the bug analysis field, which will then have data on Ansible bugs.

In order to answer the research question, bug analysis must be performed on the Ansible configuration management system. After performing the steps in the methodology stage for setting up the environment, the bug analysis itself will follow suit. To analyze the bugs with the goal of extracting valuable analytic data from them, there must first be an abundance of data for each issue. A minimum of the bug description and any other artefacts such as stack traces are required, as well as the fix itself.

When answering each of the research sub-questions, categories will be made for the answers. They will be used to classify the answers into a set amount of values, which will aid in analysis. After answering all of the questions for around a 100 sampled bugs, summaries will be possible to construct. It is from these summaries and visualizations

that common patterns and useful insights can be extracted, which is the ultimate goal of this study. Using these materials, we will see the main causes of bugs for Ansible, and how they can be quicker detected and fixed. It will also give insight to which changes should be vetted more thoroughly, before merging them into the main branch, as well as other observations that can be used to derive recommendations to improve the overall code quality of the Ansible code base.

4 Results

After analyzing 100 bugs from the the Ansible configuration management system, these are the results.

4.1 Bug Classification

This sub-section contains the classifications for all of the research questions that were discovered during the analysis of the bugs.

4.1.1 RQ1: What are the common patterns of Ansible bugs?

The answer to this question are the common classifications found in the this section, as well as the findings in sub-section 4.2, where the results are analyzed.

4.1.2 RQ2: What are the main symptoms of common Ansible bugs?

A symptom of a bug is an effect the user experiences that differs from expectations. This is how the bug is noticed and reported to the bug tracker. The symptoms can be extracted from the "EXPECTED RESULTS" and "ACTUAL RESULTS" sections in the Ansible bug report format. During this study, five parent categories of bug symptoms were identified: *Unexpected Runtime Behavior*, *Misleading Report*, *Unexpected Dependency Behavior Error*, *Performance Issue*, *Crash*. There are also sub-categories, which will be mentioned in text, but do not appear in the graph for the sake of clarity.

4.1.2.1 Unexpected Runtime Behavior (URB) A bug that is classified under this symptom causes an unexpected effect during the runtime of the configuration management system. An example would be when the user notices the target host is incorrectly configured. There may be errors associated with this category, however they must be handled correctly and presented to the user in a clear and readable manner. This category was the leading symptom for Ansible bugs in this analysis.

Configuration does not parse as expected (URBCDNP) This is a sub-category of the *Unexpected Runtime Behavior* parent symptom category, and it is used to classify bugs that caused the the SUT[1] to incorrectly parse the configuration. This would result in the system executing the configuration like normal, however the internal representation was incorrect, and thus the system behaved erroneously.

Target misconfiguration (URBTM) This is another sub-category of the *Unexpected Runtime Behavior* parent category. It is used when the target was configured incorrectly due to a bug, which caused the user to notice the bug.

4.1.2.2 Misleading Report (MR) This symptom category was used to categorize bugs that caused the SUT to return incorrect reports to the user. An example would be a bug due to which the SUT returns a status report claiming the configuration of the remote host has failed, when in reality it was successful.

4.1.2.3 Unexpected Dependency Behavior Error (UDBE) A bug classified under this category manifests as an incorrect behaviour of a dependency of the system. This could be a scenario where a dependency such as Docker is performing not as expected, due to a bug in the SUT.

4.1.2.4 Performance Issue (PI) A bug with the Performance Issue symptom classification manifests when the system performance drops and negatively affects the user experience.

[1] System Under Test

4.1.2.5 Crash (C) This symptom parent category is used to classify bugs that cause the SUT to crash when executing. Ansible is a modular system, where there is a core and the rest of functionality is implemented via modules, which have their errors handled by the core, which incurs a 'soft crash', where the core peacefully halts execution. However, if the error returned from the module is not formatted, or is just a stack trace, then it is classified as a crash, since nothing handled the error apart from the core safety catch. In addition, Ansible playbooks mainly work imperatively, and such a failure halts execution, meaning all subsequent tasks would not be executed, unless this behaviour is manually overridden. This scenario constitutes a crash instead of an *Unexpected Runtime Behavior (URB)*.

Feature/sub-feature non functional (CFNF) This is a sub-category of *Crash (C)*. A bug with this symptom causes a feature or a part of a feature to become non functional, crashing the configuration execution flow in the process.

Execution crash (CEC) This is another sub-category of *Crash*. Bugs that were classified to this symptom category caused the SUT to crash during execution due to a fault in the core of Ansible.

Configuration parsing crash (CCP) This is the third sub-category of *Crash*. If the SUT crashes while parsing the configuration passed to it by the user, the bug that caused the crash would be categorized under *Configuration parsing crash*. This classification is limited to when the crash happens during the parsing of the input.

4.1.3 RQ3. What are the main root causes of common Ansible bugs?

A root cause of a bug determines what part of the system ultimately failed due to the bug, and produced the symptoms discussed above. The root cause was extracted from the fix, since it is usually local to an area of the software that caused the bug. Four parent categories of Ansible bug root causes were identified: *Error Handling Reporting Bugs*, *Misconfiguration inside the codebase* , *Target machine operations*, *Controller machine operations*. There are some sub-categories, which are omitted from the graphs.

4.1.3.1 Error Handling Reporting Bugs (EHRB) A bug that is classified under this root cause category is caused by the error handling and reporting components of the configuration management system. This classification is specific to components created specifically for handling or reporting errors.

4.1.3.2 Misconfiguration inside the codebase (MC) This root cause parent category houses the bugs that originate from default value misconfiguration or dependency misconfiguration inside the code base itself.

Misconfiguration of default values inside the codebase (MCDV) This is a sub-category of *Misconfiguration inside the codebase*. For a bug to be classified under this category, the default values inside the code base must be the root cause of the bug.

Misconfiguration of dependencies inside the codebase (MCDP) This is another sub-category of *Misconfiguration inside the codebase*. A bug that occurs because the dependencies or their versions were incorrectly configured inside the database is classified under this root cause category.

4.1.3.3 Target machine operations (TMO) This is a parent root cause category, and it is assigned when the root cause of a bug has to do with the area of operations on the target host.

Target machine incorrect filesystem operations (TMOFS) If a bug is caused by mis-managed file system operations on the host, such as file permission and ownership setting or similar, the bug's root cause is classified under this category.

Target machine / remote host has dependency issues (TMOD) A bug is classified under this root cause if it is caused by the remote host having unmet or misconfigured dependencies.

Fetch target machine variable/facts failure (TMOFTMF) If collecting information from the remote host causes the bug, it is classified under this category.

Target machine parsing issue (TMOPI) In the case where a bug prevents data from being parsed correctly on the target machine, this category is used.

Target machine instruction translation error / Abstraction layer error (TMOITE) In the case where a bug prevents data from being parsed correctly on the target machine, this category is used.

4.1.3.4 Controller machine operations (CMO) This is a parent root cause category, and it is assigned when the root cause of a bug has to do with the area of operations on the controlling machine where Ansible is running.

Controller machine executor has problems (CMOEP) This sub-category is used when the executor experiences issues on the controlling machine.

Controller machine connection has problems (CMOCONP) If a bug causes the connection to fail or become unstable from the controller machine to the target machine, this category is used.

Controller machine parsing issue (CMOPI) If a bug prevents data from being parsed correctly on the controlling machine, the bug is classified under this category.

4.1.4 RQ4. What is the impact of common Ansible bugs?

4.1.4.1 Impact Severity The first severity level of an issue is **Low**, if the bug affects an edge case use of a sub-feature of the system, but the system works overall. The second severity level of an issue is **Medium**, if the bug makes the system fail on an important use case, but works otherwise. The third severity level of an issue is **High**, if the bug makes the system fail on multiple important tasks.

4.1.4.2 Impact Consequences The impact consequences of a bug describe what was the end result of the problem the bug caused.

Performance degradation (PD)
When the performance is decreased due to a bug.
Logs reporting failure (LOGRF)
When logs fail to be reported due to a bug.
Target configuration failed (TCF)
When the target configuration fails. The system may or may not report a crash.
Target CMS operation crash (TCFC)
When the CMS crashes while preparing actions on the target machine.
Target configuration inaccurate (TCIA)
When the target configuration succeeds, but is not accurate to what is desired.
Target configuration incomplete (TCIN)
When the target configuration succeeds, but only partially, and multiple tasks are not executed, leaving the target machine configuration incomplete.
Confusing user experience (CUX)
When the configuration management system reports conflicting data to the user, or if the user attempts to use a feature and receives a completely unexpected result. This also covers cases where the sample implementation is incorrect and misleads the user.

4.1.5 RQ5. What are the fixes of common Ansible bugs?

The fixes of a bug are categorized in two ways. The first one is an actual code change, the second one is the conceptual component change of the software.

4.1.5.1 Code Fixes There are a set of classifications for the code fixes of a bug. They are: **Change on data declaration/initialization (CDDI)** when a data declaration is changed, **Change on assignment statements (CAS)** when assignment statements are changed, **Add class (AC)** when a class is added, **Remove class (RC)** when a class is removed, **Change Class (CC)** when the contents of a class are changed, **Add method (AM)** when a method is added, **Remove method (RM)** when a method is removed, **Change method (CM)** when the content of one or multiple methods are changed, **Change loop statements (CLS)** when there is a fix regarding a loop statement, **Change branch statements (CBS)** when there is a fix about core branches, **Change return**

statement (CRS) when one or more return statements are changed, and **Invoke a method (IM)** when a new method has been invoked.

4.1.5.2 Conceptual Fixes .

The conceptual fixes are broken down into Execution component changes (**Fix execution component (FEC)**, **Expand execution feature (EEF)**), Parser component changes (**Fix parser component (FPC)**, **Expand parser feature (EPF)**), Connectivity component changes (**Fix connectivity component (FCC)**, **Expand connectivity feature (ECF)**), Dependency changes (**Change dependencies (CDEP)**), System structure changes (**Change system structure (CSS)**), Configuration changes (**Change configuration (CCONF)**), and changes to diagnostic messages displayed to the user (**Displaying a diagnostic message to the user (DDM)**).

4.1.6 RQ6. Are Ansible bugs system dependent?

Each bug is either dependent on a system or not. This includes both the scenario when Ansible crashes because it is running on a particular operating system, and also the scenario when the target machine has to be a particular kind of system in order for the bug to appear.

The answer is either **True** or **False**, and the systems which the bugs depend on are also tracked for analysis purposes.

4.1.7 RQ7. What are the triggers of common Ansible bugs?

The triggers are split into the trigger cause (what introduced the bug) and reproduction trigger (how can the bug be reproduced).

4.1.7.1 Trigger Cause .

Trigger causes are split into four categories. **Logic Errors (LE)** are the cause when the error is introduced by a contributor by setting up faulty logic for branch statements, missing switch cases, or misusing programming language features. **Algorithmic Errors (AE)** are the cause when a contributor incorrectly implements an algorithm, uses the wrong algorithm, or when the effect of a function implementation is not what it should be. **Configuration Errors (CE)** are the cause when a contributor incorrectly configures various values and dependencies inside the code base. **Programming Errors (PE)** are the cause when the contributor fails to oversee various edge cases such as non-existing array index access or an exception-prone operation not being wrapped in try-catch, which leaves the error unhandled.

4.1.7.2 Trigger Reproduction

Trigger reproduction covers the method by which the bug can be reproduced.

CLI commands (CLIC) .

The bug can be triggered by executing a feature of Ansible via CLI. A sub-category of this is **CLICDMO**, which covers dependency module operations via CLI.

Environment setup (ENVS).

The bug can be triggered by setting up the environment in a certain way. This includes target machine setup and environmental variable setup.

Faulty Dependency Usage (FDEPU).

The bug can be triggered by incorrectly utilizing a dependency of Ansible.

OS specific execution (OSSE).

The bug can be triggered if the host or the target is running a specific kind of operating system, allowing the bug to surface.

Test case (TC).

The bug can be triggered by running a test case on a snapshot of the code base prior to the fix being implemented.

Specific Invocation (SI).

The bug can be triggered by invoking the software management system with a specific configuration. The sub-categories are: **Target machine control execution (SITMCE)**, when the specific invocation involves target machine control, **Internal module invocation (SIIMI)**, when an internal module is invoked via a playbook task, **Custom module invocation (SICMI)**, when the specific invocation involves a custom module that the user made, **Config/Runbook Parsing (SICRP)**, when Ansible has to be invoked with a specific run book to reproduce the bug.

4.2 Analysis Results

This section contains the actual findings using the bug attribute classifications described in section 4.1. The bug attribute classes described in 4.1 and the upcoming findings answer **RQ1: What are the common patterns of Ansible bugs**, since they contain insights on the common patterns of all of the attributes of a bug.

4.2.1 RQ2: What are the main symptoms of common Ansible bugs?

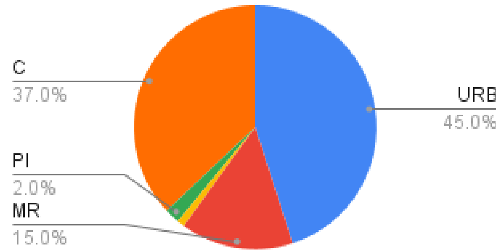

Figure 1: Symptom distribution over the analyzed bugs.

Figure 1 shows the distribution of symptoms over the set of analyzed Ansible bugs. The leading symptom for Ansible bugs was unexpected runtime behaviour, with 45% of the analyzed bugs being categorized under this symptom. The second largest contributing symptom is the crash parent category, which constitutes 37% of all of the classified bugs. Trailing behind are the misleading report, performance issue, and the unexpected dependency behavior error comprising 15%, 2%, and 1% of all bugs respectively.

This distribution reveals that Ansible suffers most from behaviours that do not cause crashes of the core or the modules, but instead bug affects the system in a way that produces some unwanted effect while executing the configuration. These types of issues are difficult to catch since every code path and side effect of the execution needs to be checked in order to identify them. One potential way to more thoroughly test vulnerable code paths is a form of white-box testing that leverages genetic algorithms to extract the most valuable test cases for maximizing statement, branch and loop coverage, minimizing potential future bugs in the process [7].

In addition, the bugs classified under misleading report are another possible area of focus, as these types of bugs have to do with the messages displayed to the user. Possible vulnerabilities of this nature come about due to ineffective wording and similar output-related oversight. These vulnerable areas could be statically detected due to their footprint on the logger, and this could be used to detect and fix such bugs before they ever manifest.

4.2.2 RQ3. What are the main root causes of common Ansible bugs?

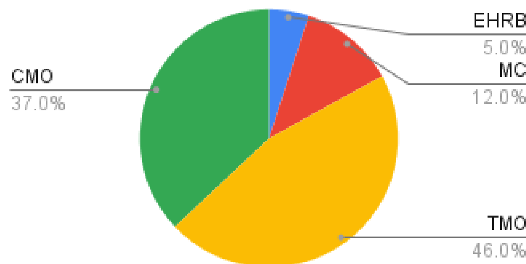

Figure 2: Root cause distribution over the analyzed bugs.

Figure 2 displays the distribution of root causes over the analyzed Ansible bugs. The predominant root cause for the set of analyzed bugs were target machine operations, at 46%. This can be attributed to the fact that the majority of the code in the code base is directly related to operating on the target machine, since performing configuration functions on the main purpose of Ansible. The second biggest cause are the control machine operations, at 37%. This

is understandable, due to the fact that Ansible is an agent-less configuration management system, and thus a large amount of the processing required to configure the host system happens on the controlling machine. The last two root causes are the code base misconfiguration category at 12% and the error handling reporting category at 5%.

To address these root causes, the maintainers could use this distribution data to identify the areas of the code base that need more vetting when it comes to merging in changes. This could take place in the form of additional review requirements for pull requests modifying components interfacing with the target machine, for example. Additionally, for any form of periodic testing, specific target areas could be prioritized, such as allocating more run time for configuration fuzzing tests for that component, to more effectively prioritize finding bugs in the more problematic areas.

4.2.3 RQ4. What is the impact of common Ansible bugs?

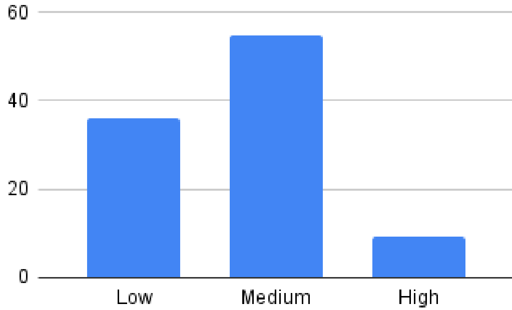

Figure 3: Impact severity distribution over the analyzed bugs.

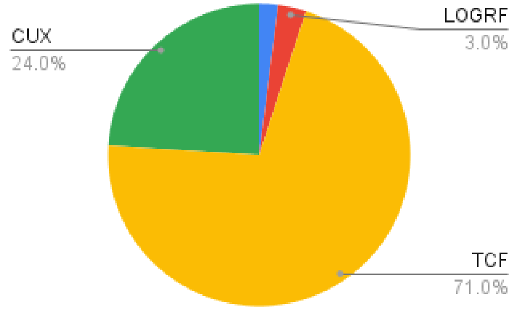

Figure 4: Impact consequence distribution over the analyzed bugs.

Figures 3 and 4 show the distributions of impact severities and impact consequences for the analyzed Ansible bugs.

Starting with the severities of the bugs, the majority of the bugs are classified as medium severity bugs, at 55%. The second severity by amount of bugs is low, covering 36% of all analyzed bugs. Finally, the high severity is last in terms of number of occurrences, with 9% of the bugs classified as such.

Moving on to the impact consequences of the bugs, the leading category was target configuration failed, comprising 71% of the analyzed bugs. The second largest contributor was confusing user experience at 24%. Then follow log reporting failures and performance degradations at 3% and 2% respectively.

Looking more closely at the leading category, target configuration failure is the leading consequence of a bug. This is expected, since as mentioned before, setting up a remote host machine given a set of configuration is the primary purpose of Ansible, which also leaves it the most vulnerable in terms of bug impact. The second most popular category, confusing user experience, correlates with the misleading report symptom category, and could be addressed by following the recommendations laid out in section 4.2.1

4.2.4 RQ5. What are the fixes of common Ansible bugs?

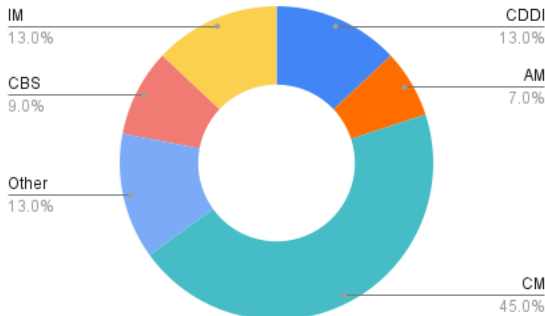

Figure 5: Code fix distribution over the analyzed bugs.

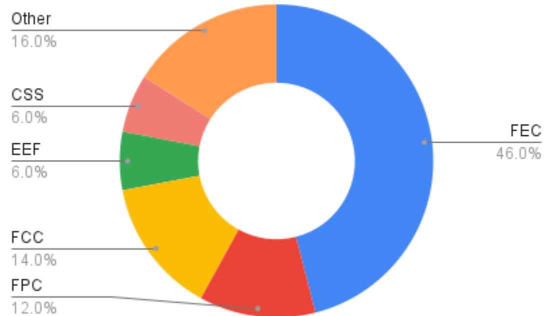

Figure 6: Conceptual fix distribution over the analyzed bugs.

Figures 5 and 6 display the distributions of the code fixes and conceptual fixes for the set of the analyzed bugs.

In terms of the code fixes, the leading fix was change method, which encompasses changes to the body of one or multiple methods within the code base, comprising 45% of all analyzed bugs. In the second place there is a tie between the 'Change on data declaration/initialization' and 'Invoke a method' fixes, each clocking in at 13%. Following them, the change branch statement fix covers 8%, and the add method fix covers 7% of all of the cases. The rest of the code fix categories combined account for the remaining 13%.

Switching now to conceptual fixes, an overwhelming majority of fixes of the bugs had to do with fixing an executor component, at 46%. The second largest conceptual fixes were to connectivity components at 14%, third are fixes to parsing components at 12%. Following this there is a tie of 'Expansion of execution feature' and 'Change system structure', both at 6% of the total Ansible bugs analyzed. The remaining categories make up the remaining 16%.

Upon examination, it seems evident that the execution components are the ones most in need of fixing, which makes sense considering they comprise the bulk of the code base and are responsible for actually completing the primary purpose of Ansible. However, one notable aspect is that the second most important category, fixes to connectivity components, is relatively high up the rankings because of the fact that Ansible is agent-less, and therefore requires solid infrastructure for connecting to the host machine, since the commands for each task in a runbook need to be transmitted to the target prior to execution.

4.2.5 RQ6. Are Ansible bugs system dependent?

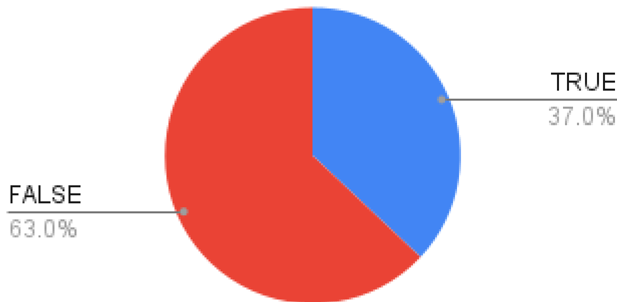

Figure 7: System dependence distribution over the analyzed bugs.

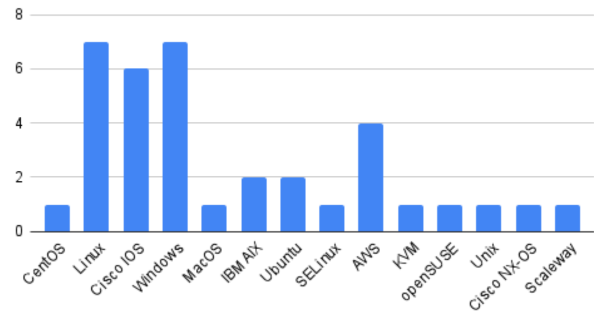

Figure 8: Dependent system distribution over the analyzed bugs.

Figures 7 and 8 display the distributions of the system dependence and the dependent systems over the bugs that were analyzed in this study.

Overall, around 63% of all bugs are system independent, and 37% depend on some type of system to manifest. This distribution is sensible, because most of the execution tasks in Ansible are run inside a Python abstraction layer, thus a lot of the operations are system agnostic.

Taking a look at the distribution of dependent systems that comprise the 37% of bugs, Linux and Windows tie for 7% of all system dependent bugs, with Cisco IOS (Cisco Internetwork Operating System) coming in at a close second at 6% [11]. After that, the next notable system dependency is AWS at 4% of the analyzed bugs. This consists of various AWS-specific functionality, that has to be executed within AWS for the bugs to appear. The next two entries of IBM AIX [12] and Ubuntu tie at 2% each of the total bugs assessed. The rest of the systems make up the remaining 9% of the bugs.

An interesting observation is that since Ansible supports a wide array of functionality for lots of different types of systems, bugs seem to surface for niche systems such as Cisco IOS. From this, one can derive a possible conclusion that in order to decrease the amount of bugs that manifest on specific systems, local platform testing could be employed. For this approach, environments with the requisite system would be spun up for the test executors, and the priority of the systems to test could be taken from the data discovered here. Alternatively, if there is no way to set up a local instance of a system, such as the case of AWS, an integrated sandbox environment could be employed to perform the testing in [13].

4.2.6 RQ7. What are the triggers of common Ansible bugs?

Figures 9 and 10 show the distributions of the trigger causes and trigger reproductions over the analyzed bugs.

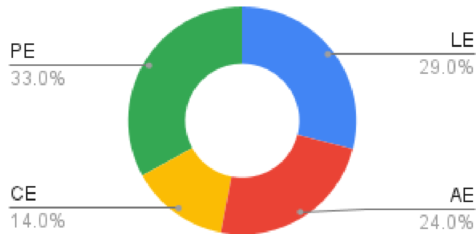

Figure 9: Trigger cause distribution over the analyzed bugs.

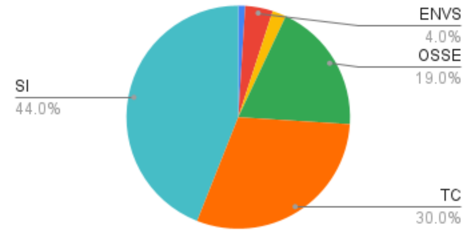

Figure 10: Trigger reproduction distribution over the analyzed bugs.

For the trigger causes, the distribution is relatively similar over the three leading causes of programming errors, logic errors, and algorithmic errors at 33%, 29% and 24% respectively, with configuration errors falling behind at 14%. This is because configuration errors are in general more rare, since they can be spotted more easily when reviewing code.

Switching over to the trigger reproduction of the bugs, specific invocation is the leading reproduction trigger with 44% of the bugs covered, seconded by test case invocation at 30%. Following up, there is OS-specific execution with 19% of all bugs, and environment setup with 4%, and the remaining categories comprise the remaining 3% of the bugs analyzed.

Observing the fact that specific invocation is the leading reproduction trigger, one possible approach to explore the bugs of this class could be a system that leverages past bug behaviour in order to perform fuzzing over various inputs that trigger potentially vulnerable code paths, which would reduce the amount of untested code paths and thus decrease the occurrences of such bugs [8].

5 Responsible Research

In this section, possible ethical issues as well as the reproducibility of the results will be discussed.

5.1 Possible Ethical Issues

During the course of the research, there is a possibility for infringements of ethics norms to come about. It is the researcher's duty to correctly identify, evaluate, and avoid such issues, in order to keep the study ethical. In the case of this study, there are two main ethical issues that need to be addressed.

The first ethical issue concerns the publishing and use of other people's work, namely the bug reports in the from of GitHub issues that have been filed by users of Ansible, and the bug fixes in the form of GitHub pull requests that have been submitted by the contributors of Ansible. Both issues and pull requests require an agent to formulate them, requiring a degree of creativity, which makes them a form of intellectual property (IP). To ensure no rights are being violated in the course of this study, the license of Ansible must be consulted first. Ansible is licensed under the GNU General Public License v3.0 [14]. The important part of the license in resolving the ethical concerns for this study is regarding derivative works. The use of the software is allowed for any private or commercial entity, and the source can be modified without publishing the altered code for private, non-commercial use. However, for distributed and/or commercial use, the modified code must be published, and Copyleft applies to the derivative product, from the same license, in order for end users to retain the same freedoms offered by the original license. Thus, all contributions all bug reports and bug fixes produce derivative versions of Ansible that are free to use for any purpose by any private or commercial entity, which covers the use case of this study.

The second ethical issue under examination is the use of GitHub API in order to retrieve the bugs for examination. This has the potential to negatively affect the load levels of the GitHub API, resulting in increased compute resource use for GitHub and in potentially degraded service availability for other users of GitHub services. This would make the study slightly unethical, because it would incur a negative effect on the experience of other users of GitHub. To mitigate this risk, a custom application was registered and an API key was requested from GitHub for that application. This makes the impact of the bug data requests required for this study track-able, and makes it so limits can be imposed from GitHub's side. By setting up the API access path in this way, the study operates by GitHub's rules, since they have a set of rate limits for custom applications [15], which includes the application registered for this study. GitHub is free to monitor and adjust limits or shut down the access to the application, if the service impact is too great,

by GitHub's own evaluations, thus the GitHub API usage levels required for this study were acceptable, and as a consequence, this aspect has no ethical impact on the study.

5.2 Reproducibility

In order to make sure a study is verifiable, it must first be reproducible. This study was completed following the strategy established in section 2, and a concrete setup plan has been described in methodology implementation in section 2.3. The study can be reproduced by following the steps laid out in the methodology section of the paper and clarified in the implementation sub-section of the methodology. In addition, all of the tooling that was created for completing this study is available publicly on GitHub [16]. This ensures that it is possible for any reader of the study to replicate the results, since the implementation is already complete, and someone interested in reproducing the results of the study can choose to use the provided code and simply execute the data collection and processing steps as is.

One notable factor that needs to be addressed is the sampling aspect of the study. The Ansible GitHub repository contains 3500+ bugs that are of the correct type as outlined in the methodology section, and have valid fixes. Because of this fact, it not possible to thoroughly analyze all of them in a single study. This elicited the use of a sampling technique for this study, which uses random operations to pick out a subset of the issues for analysis. This study addressed 100 bugs, and it is likely that a reproduction study would result in a different set of bugs being selected for analysis, even if the methodology was followed exactly. One possible effect of this are slightly different findings in the reproduction study, however this is an unavoidable side-effect of the sampling methodology. However, an ample sample size was selected to ensure that even if the study is reproduced with a different set of issues, comparable trends should appear in the results, and thus the findings as well as the conclusions should not differ by a large margin from this study.

6 Discussion

In this section, the findings will be discussed and the threats to validity will be addressed.

6.1 Insights

From the findings, a clear picture can be formed about the common characteristics of Ansible bugs. After examining 100 Ansible bugs, common classifications became apparent for each of the research questions, which enabled the questions to be efficiently answered for each bug. The most predominant symptom of Ansible bugs was unexpected runtime behaviour, meanwhile the most common root cause was target machine operations. Most of Ansible bugs are of medium impact, and they most often result in the failure to configure the target host. Ansible bugs are usually fixed by changes contained to the body of a method, and they predominantly fix execution components. Ansible bugs are not very system dependent, however from the bugs that are system dependent, Linux and Windows are the most bug-prone platforms. The triggers of Ansible bugs are mostly programming errors and logic errors, meanwhile the main reproduction trigger is specific invocation of Ansible. An important factor to note is that since most bugs appear in the execution and connectivity components of Ansible, investing more time in testing them would be of great benefit to the code quality of the software. Moreover, the bugs are usually triggered by specific invocation of Ansible, meaning input fuzz testing would also provide a large positive impact as well.

The findings can be utilized in future studies for developing various testing techniques for Ansible and other configuration management systems. The proposed areas of exploration for increasing the code quality of the code base comprise of a multitude of possible new test methodologies. Tests using genetic algorithms could be used to greatly improve the test coverage of the project and of new incoming code in the form of bug fixes [7]. On-demand OS-specific environment test procedures could be investigated to reduce the amount of system-dependent bugs. And lastly, utilizing past bug behaviour outlined in this study could be a great pathway to perform effective fuzz testing on the various specific input cases that constitute the majority of reproduction triggers of Ansible bugs [8].

An important observation from the findings is that around 30% of the bug fixes found in the Ansible code base contained one or more associated test cases. The proportion found is influenced by the choice to not limit the time period of the bug sampling, however the fact that the coverage for incoming bug fixes is quite low still stands. This is an issue, since it significantly increases the difficulty of reproducing Ansible bugs, and it also introduces a high risk of regression if a future change breaks a previous fix. The recommendation here is that the Ansible team 1) should expand the test case requirements of incoming new code and 2) should enact much stricter checks on vulnerable code base paths, which can be determined from the findings of this study. That should in effect greatly reduce the amount of incoming bugs and regressions that are usually missed.

6.2 Threats to Validity

This sub-section will address possible shortfalls of this study method.

Firstly, the bugs analyzed should be representative of the experience of an end user of the Ansible configuration management system. This is to ensure that this analysis is valid in terms of how users interact with the product. The bugs should not be inconsequential to the functionality of the software, so documentation bugs are excluded from the study. An addition, only the bugs that have been already addressed are of interest, as the ones that do not yet have a solution are mutable, and can be re-categorized or closed for being inaccurate, so they should be excluded. Regarding bugs with fixes, only the bugs that have their fixes merged into the code base should be sampled, since the non merged fixes can still be subject to scrutiny from the maintainers and they may not be valid.

Secondly, only an officially recognized source for bugs should be consulted, to ensure the correctness of the data being analyzed. Ansible recognizes GitHub issues as the official bug tracker for Ansible [9], so that is what this study used.

Thirdly, the maturing timeline of the Ansible configuration management system must be addressed. The sampling method of this study resulted in bugs being taken from various points in time, from near the very beginning of Ansible to current time. It is important to recognize that the practices and code quality within the code base has evolved over that period of time, an example of this is that tests become more frequent the more recent the bug fixes are. This is a side-effect of the completely random sampling of bugs, and in future studies it may be worthwhile to invest in time-bounded sampling of Ansible bugs.

7 Related Work

This section will discuss the work that is related to the content of this study.

There is a limited amount of work regarding specifically the bugs within the code base of the Ansible configuration management system, which was part of the main motivation for this study.

A notable study that acted as a methodological base for this study is the Study of Typing-Related Bugs in JVM Compilers by Chaliasos et al. (2021) [17]. This work contains a solid outline of what a bug study should be, and it has been used as inspiration for sections of this study. This study covers a larger amount of bugs, namely 320, which lends itself to improved accuracy of the results of that study. In addition, this study covers a multitude of different compilers, which offers great utility in deriving contrasts between different systems, and determining which possible paths for improvement are suitable for each one.

Another paper about Ansible by M. Hassan et al. (2022) contains an in-depth examination of bugs found within the user-provided Ansible scripts found in open source repositories is also related to the current study [4]. This research focuses on content created by non-contributors of the software, which is a different focus from this study, and could be utilized to further expand the field of Ansible testing. It is discovered that 1.8% of the sampled publicly hosted Ansible test scripts contain a bug, and 45.2% of the tested 104 public repositories contain at least one bug in their test scripts.

Continuing with Ansible related studies, there is another study that investigates the security-related code smells of public Ansible scripts by Rahman et al. [5]. This study expands on the security issues found in 50323 public scripts of Ansible found within 813 public repositories. The examined scripts include 46600 instances of security smells, and of those, 7849 issues are regarding hard-coded passwords. This study focuses on the human created content in the form of scripts, and thus does not reflect the state of the code quality within the code base itself. However, this is a good avenue to continue exploring, in order to further the landscape for improving the software quality of Ansible.

8 Conclusions and Future Work

This study has achieved it's goal of bringing forth the overall commonalities of the bugs within the Ansible code base. They have been used to extract sets of useful classifications that can be re-used in future endeavours to continue developing the bug analysis landscape of Ansible, and of other configuration management systems. Useful insights have been drawn from the findings on where the Ansible team may want to invest to improve the overall code quality of Ansible. A variety of methods for resolving the main outstanding issues for each category were proposed, from genetic algorithm testing [7] to OS-specific environment emulation and specific vulnerable input fuzzing [8]. These proposals can be further expanded upon by future studies.

The execution and connectivity components of Ansible were outlined as being the most bug-prone in the examined data, and a recommendation is made to invest more time in testing, as that has the highest probability to improve the code quality of the software. In addition, Ansible bugs are mostly triggered by specific invocation of Ansible, and as

such a recommendation of incorporating input fuzz testing is made on the basis that it would also be of high impact for the quality of the software.

One possible area for future improvement is to more deeply examine which exact areas caused which crash, which could allow the development of specialized tools to more thoroughly analyze possible problematic changes to the code before they are merged into the code base. This would also be of great benefit to the fuzz testing methodology referenced in this study, as it would function better with more information of past behaviour of bugs [8].

Another possible area for research is to identify a more concrete set of trigger causes for the bugs, since this study focused in on four broad areas of programming errors, logic errors, algorithmic errors, and configuration errors. By extracting a hierarchical structure of the trigger causes of bugs of the Ansible configuration management system, a heuristic model of the most vulnerable contributor changes could be rendered. This would open the doors to creating Ansible-specific sophisticated methods of testing that could effectively identify possible problem areas in pull requests, wherein a more in-depth review procedure could be enacted for such changes.

9 Acknowledgements

I would like to thank the team of my research project group, Bryan He, Mykolas Krupauskas and Mattia Bonfanti, for peer reviews and productive discussions during the course of the study. Thank you to Mykolas Krupauskas for creating a web UI for quick analysis of issues, which helped speed up the progress of the research. Finally, thank you to our supervisor Theodoris Sotiropoulos and our supervising professor Diomidis Spinellis for the continued support and invaluable feedback.

References

[1] Domenico Cotroneo et al. "How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation". In: *Journal of Systems and Software* 113 (2016), pp. 27–43. DOI: 10.1016/j.jss.2015.11.021.

[2] Rongxin Wu et al. "ReLink". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11* (2011). DOI: 10.1145/2025113.2025120.

[3] Red Hat Ansible. *Ansible is Simple IT Automation.* 2022. URL: https://www.ansible.com/?hsLang=en-us.

[4] M. Hassan and Rahman. "As Code Testing: Characterizing Test Quality in Open Source Ansible Development". In: *ICST 2022 Research Papers* (2022). URL: https://akondrahman.github.io/files/papers/icst2022-tama.pdf.

[5] Akond Rahman et al. "Security Smells in Ansible and Chef Scripts". In: *ACM Transactions on Software Engineering and Methodology* 30 (1 Jan. 2021), pp. 1–31. ISSN: 1049-331X. DOI: 10.1145/3408897.

[6] Mehdi Bagherzadeh et al. "Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–32. DOI: 10.1145/3428282.

[7] Rizal Broer Bahaweres et al. "Analysis of statement branch and loop coverage in software testing with genetic algorithm". In: *2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)* (2017). DOI: 10.1109/eecsi.2017.8239088.

[8] Sanjeev Das et al. "A Flexible Framework for Expediting Bug Finding by Leveraging Past (Mis-)Behavior to Discover New Bugs". In: *Annual Computer Security Applications Conference* (2020). DOI: 10.1145/3427228.3427269.

[9] Red Hat, Inc. *Reporting Bugs And Requesting Features â Ansible Documentation.* 2017. URL: https://docs.ansible.com/ansible/2.3/community/reporting_bugs_and_features.html.

[10] Santiago Dueñas et al. "Perceval: software project data at your will". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings.* ACM. 2018, pp. 1–4.

[11] Cisco Systems, Inc. *Cisco IOS.* Sept. 2020. URL: https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-software-releases-listing.html.

[12] International Business Machines Corporation. *AIX, UNIX for IBM Power Systems.* 2022. URL: https://www.ibm.com/it-infrastructure/power/os/aix.

[13] Amazon.com, Inc. *Best practices for creating and managing sandbox accounts in AWS.* Feb. 2021. URL: https://aws.amazon.com/blogs/mt/best-practices-creating-managing-sandbox-accounts-aws/.

[14] Free Software Foundation, Inc. *The GNU General Public License v3.0*. June 2007. URL: `https://www.gnu.org/licenses/gpl-3.0.en.html`.

[15] GitHub, Inc. *Rate limits for GitHub Apps*. 2022. URL: `https://docs.github.com/en/developers/apps/building-github-apps/rate-limits-for-github-apps`.

[16] Matas Rastenis. *tudelft-research/ansible*. June 2022. URL: `https://github.com/MKrupauskas/tudelft-research/tree/main/ansible`.

[17] Stefanos Chaliasos et al. "Well-typed programs can go wrong: a study of typing-related bugs in JVM compilers". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pp. 1–30. DOI: `10.1145/3485500`.