ABSTRACT

HORTON, ERIC. Automatic Inference of Computing Environments . (Under the direction of Chris Parnin.)

Software developers are increasingly responsible for the day-to-day deployment and management of applications, a practice known as DevOps (a portmanteau of development + operations). One key component of DevOps is the practice of software configuration management, where a computing environment is altered so that it supports program execution. Configuration management is necessary because most programs have implicit expectations about the structure of the computing environment and the resources available to them at runtime, a fact that also makes correct configuration management both difficult and time consuming. And when developers get configuration wrong, the results can include outages, loss of customer data, and financial ruin.

The purpose of this work is to investigate specific challenges that developers face while performing configuration management and create a suite of automated tools that reduce the requisite knowledge and effort from developers to configure an environment. We begin with a corpus of 10k simple Python programs, finding that 75% encounter a failure when run without any configuration. When developers were asked to produce a working configuration for these programs, they reported difficulty in finding or installing dependencies, incorrect versions, missing system libraries, and more.

We address the problem of dependencies by introducing DockerizeMe, a tool for automatically producing a Dockerfile that includes dependency configuration using a combination of static analysis, dynamic analysis, and association rule mining. DockerizeMe was shown to resolve dependency related errors for many of the Python programs from our corpus. A subsequent tool, V2 aids developers in finding the correct dependency version with efficient search algorithms that incorporate prior knowledge about build breakages related to dependency version changes. V2 was shown to reduce the time it took to resolve dependency version related errors.

Finally, we address challenges with translating and writing configuration scripts though Dozer and Synth. Dozer uses techniques for tracing and comparing system calls to find similar configuration tasks in multiple configuration languages. These comparison results can be used to automatically migrate the configuration task from one language to another. Synth focuses on synthesizing full configuration scripts by treating configuration as a covering problem where a set of desired changes must be reproduced by combining the changes of one or more configuration tasks. Our results show that Synth can produce working configuration scripts for many environments.

Automatic Inference of Computing Environments

by
Eric Horton

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2022

APPROVED BY:

John-Paul Ore	Brad Reaves
Kathryn Stolee	Chris Parnin Chair of Advisory Committee

ACKNOWLEDGEMENTS

I would like to thank Chris for his support and advice over the last several years. The majority of this work would not have been possible but for his guidance. Additional thanks to the rest of the Alt-Code lab who made efforts to support and help one another while working from home during the middle of a pandemic. The last two years of grad school would have been far more isolating and terrifying if not for your presence. And finally, thank you to Dr. Blair Sullivan, who changed my life on three separate occasions with an email, an offer, and an introduction.

Funding

This work is funded in part by the NSF SHF grant #1814798.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Software configuration management is the process of changing a computing environment to enable the successful execution of a program by carrying out one or more configuration tasks [Hor18]. As a practice, configuration management is necessary because even simple programs make assumptions about the state of the system and the resources available to them at runtime [Seo14; Yan16; Hor18; Hor19a; Hor19b]. They rely on these assumptions in order to run correctly, and produce crashes or other unexpected behavior when those assumptions are not met.

The impacts of incorrect configuration management can be severe. It's estimated that the average hourly cost of infrastructure failure is $100K and can cost a Fortune 1000 company up to $1.25-2.5B per year [Idc15]. Notable outages related to configuration include a site-wide blackout for Facebook,[1] service failure at Netflix,[2] and the loss of all external connectivity on Google's cloud platform.[3] In more extreme cases, incorrect configuration can lead to the loss of customer data and unrecoverable financial loss. In 2017, GitLab accidentally deleted 300 GB of customer data while troubleshooting an incident with their production database. When the developers tried to restore the data from backups, they discovered that no recent backups had succeeded because the backup procedure was configured to use the wrong version of the backup utility [Git]. At Knight Capital Group, an incorrect configuration in 2012 activated an old program feature that resulted in the loss of $460M in 45 minutes, bankrupting the firm [Sev14].

There are many different types of configuration changes that a developer can make to an environment. They might install third-party libraries that a program depends on, write

[1] `https://techcrunch.com/2010/09/23/facebook-downtime/`
[2] `https://www.thestrangeloop.com/2017/antics-drift-and-chaos.html`
[3] `https://status.cloud.google.com/incident/compute/16007`

configuration files that determine how the program will run, alter file ownership and permissions, create user accounts, start services, and more [Lun10; Mac18; Ger07]. Further, these configurations are often made in different ways, even when the resulting changes are similar. For example, installing dependencies and creating user accounts are both ultimately filesystem changes, but developers usually use dedicated tools for performing each configuration rather than manually altering each file on the filesystem. The different ways in which configuration tasks can be enacted make the challenge of configuration management not just about the ability to determine which changes are required for program execution, but also about a developer's knowledge of appropriate tools and ability to apply them correctly.

Several configuration management systems exist for the express purpose of aiding developers with easy and reproducible configuration through the configuration as code paradigm [Rah18; Jia15]. Such tools include Ansible [Ans], Puppet [Pupb], and Chef [Che]. Despite having these tools, configuration management remains a non-trivial task because environment configuration is not an inherent property of code. Although this is simple to state, this observation has several implications. Program dependencies are not always obvious, and, when they can be found, the correct version is likewise difficult to track down. Many dependencies also come with their own configuration challenges, leading developers down a rabbit hole of configuration management. It is also difficult to discover services or other system states that are necessary for execution. Even when a sufficient configuration exists, it is difficult to verify that it satisfies the requirements of the program, a challenge closely related to the oracle problem for software testing [Bar15]. Overall, developers require a high level of knowledge to overcome the challenges of configuration management and produce a correct configuration.

Automated configuration techniques are faced with the same challenges as developers. Previous work has focused on creating and maintaining configuration scripts, but only by working with existing configurations or by observing developer actions [Mac18; Has18; Wei17]. The purpose of this work is to further investigate and address the challenges developers face while performing configuration management. It focuses on techniques to automatically build and validate computing environments that enable successful program execution without the need for developer input, thus reducing the requisite developer knowledge for performing configuration management.

1.1 Thesis

> Software configuration management is challenging for developers because it requires them to know about, or discover, relationships between many interconnecting components. By implementing automated inference techniques, we can reduce the amount of configuration knowledge developers need to produce correct and working environments.

1.2 Novel Contributions

1. An empirical analysis of the executability of programs hosted on GitHub.
2. A qualitative analysis of the challenges developers face while performing configuration management and how they overcome these challenges.
3. Inference techniques for determining program dependencies.
4. Search techniques for finding applicable dependency versions.
5. A language-agnostic scheme for comparing and migrating between configuration tasks from different configuration languages.
6. A framework for testing and validating changes made to computing environments.
7. A technique for synthesizing configuration scripts that produce desired changes.

1.3 Papers

1.3.1 Accepted

- Horton, E. & Parnin, C. "Gistable: Evaluating the Executability of Python Code Snippets on GitHub". *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 217–227
- Horton, E. & Parnin, C. "DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets". *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 328–338
- Horton, E. & Parnin, C. "V2: Fast Detection of Configuration Drift in Python". *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 477–488
- Horton, E. & Parnin, C. "Dozer: Migrating Shell Commands to Ansible Modules via Execution Profiling and Synthesis". *44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22)*. 2022

1.3.2 Under Review

- Horton, E. & Parnin, C. "Synthesizing Configuration Scripts". *2022 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2022

CHAPTER

2

WHY IS CONFIGURATION DIFFICULT?

This work originally published as Horton, E. & Parnin, C. "Gistable: Evaluating the Executability of Python Code Snippets on GitHub". *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 217–227

Online programming communities such as Stack Overflow and GitHub facilitate social learning of programming and API concepts. One common learning mechanism is to share code snippets or examples, which contain explanations and demonstrate how to perform a programming task or use an API [Sil12]. Code snippets are often reused and incorporated in open source projects [Yan17]. Currently, GitHub provides the ability to create and share code snippets (called *gists*), with over 300k Python gists, and over 4.5 million gists in multiple programming languages.

This work focuses on evaluating the excitability of publicly available Python scripts hosted on GitHub's gist system in the context of software configuration management (the process of configuring system environments to properly execute a software program). We seek to categorize the common reasons for why gists cannot be executed in a default environment and motivate further research on automated software configuration management by identifying what difficulties exist in properly configuring an environment to enable gist execution, specifically with regard to installing application dependencies. Our work also provides a dataset of gists known to not be executable and a baseline analysis against which to compare future configuration methods.

We start by highlighting a method for performing automated gist collection and analysis. The process involves scraping gist URLs from the GitHub gist UI. This technique led to an

initial dataset of 10,259 gists containing over 1,700 unique third-party library packages. We then cloned each gist and executed it inside of a Docker container based on the official Python image for Docker, categorizing the gist by its exit status. To evaluate gist configuration, we attempted to infer a correct environment specification using a naive algorithm that approximates the first steps human developers often take by attempting to install third-party packages by the name of the resource imported within the gist. After running the naive inference algorithm, we then reevaluated the executable status of the gist.

Our findings show that correct dependency resolution and environment configuration are often required even for small programs. Less than 25% of gists were executable by default, with over half failing due to `ImportError` in Python 2. Of the gists which initially failed with `ImportError`, our naive inference algorithm could successfully infer an environment specification less than 50% of the time.

To gain a better understanding of the why the naive inference algorithm fails, we asked 24 developers familiar with system configuration practices to create a Dockerfile for 10 unique gists, assigned to them at random, for which inference failed to resolve import errors. We then used the produced Dockerfiles and feedback from the developers to categorize gists, focusing on the first cause of failure if any gist would have failed inference due to more than one reason. The most common cause is that the names of resources used in a gist do not necessarily match the names of the packages they belong to. Gists also frequently fail due to missing transitive dependencies, missing system dependencies, configuration files, and deprecated or non-standard packages.

Finally, we present Gistable, an extensible database and framework used to perform our mining and analysis. Gistable also contains the ~5k gists with environment specifications which allow them to be run without `ImportError`. We believe that several areas of software engineering research can benefit from a database of executable code snippets, such as: automatic code summarization, testing, and API usage analysis.

2.1 Motivation

Code snippets are not always directly usable [Yan16]: They can contain parse errors or require system dependencies unmet in a programming environment. As a result, the following challenge emerges: *Given a code snippet, successfully infer the environment configuration necessary for execution.* Frequently, developers must perform this inference step manually, or rely on the creation of configuration scripts, which in itself is a time consuming task [McI11]. Unfortunately, it is not always clear what dependencies or environment configurations are required to execute code. Consider the following Python code snippet.

```
1  # Import modules from networkx and matplotlib
2  from networkx.drawing.nx_agraph import graphviz_layout
3  import matplotlib.pyplot as plot
4  import networkx as nx
5
```

```
# Generate the complete graph on five vertices
k5 = nx.complete_graph(5)

# Draw using layout generated by graphviz
plot.figure()
nx.draw(k5, graphviz_layout(k5, prog="neato"))
plot.savefig('/output/graph.png')
plot.close()
```

To successfully run this code fragment, several requirements must be met. First, the environment requires graphviz, which is a tool for visualizing graphs. Second, the environment needs to install the Python bindings for graphviz. Third, the environment needs the Python package for matplotlib and networkx. Fourth, an environment variable, `MPLBACKEND`, may be needed to specify a rendering engine that is compatible with a headless VM, which does not have a graphics display. Finally, the environment needs to ensure that an `/output` directory exists.

These requirements can also be encapsulated by a working environment configuration. One system that can be used for specifying environment configuration is the containerization system Docker. Docker configuration is centered around the Dockerfile, a configuration script which tells the Docker engine how to properly build an image that can be distributed and run by others. We present a Dockerfile for the snippet below.

```
FROM python:2.7.13
VOLUME /output
ENV MPLBACKEND Agg
RUN apt-get update
RUN apt-get install -y graphviz
RUN pip install pygraphviz
RUN pip install matplotlib
RUN pip install networkx
ADD snippet.py /snippets/
CMD python /scripts/snippet.py
```

2.2 Gistable Dataset and Tool

Gistable is a framework for collecting, evaluating and executing self-contained programming code snippets, called gists. The name is derived from a portmanteau of the words *gists* and *runnable*. Gistable is designed to support empirical research for a variety of software engineering tasks. Gistable can mine code snippets and automatically generate a Dockerfile which can be used to run the code snippet. Gistable provides a command line interface for performing tasks with the mined gists, such as checking out snippets into a working directory, and executing the code snippet inside a docker container.

```python
import requests
import json

urlbase = 'http://maps.googleapis.com/maps/api/geocode/json?sensor=false&address='
urlend = 'Zurich,Switzerland'

r = requests.get(urlbase+urlend) # request to google maps api

r=r.json()
if r.get('results'):
    for results in r.get('results'):
        latlong = results.get('geometry','').get('location','')
        latitude = latlong.get('lat','')
        longitude = latlong.get('lng','')
        break
    print latitude, longitude

else:
    print 'No results'
```

(a) Gist 10017416, which demonstrates how to use the Google Maps geocode API.

```
FROM python:2.7.13
ADD snippet.py snippet.py
RUN ["pip", "install", "requests"]
CMD ["python", "snippet.py"]
```

(b) Dockerfile containing a sufficient environment specification to run code snippet.

Figure 2.1

2.2.1 Research context

Our initial evaluation of Gistable focuses on Python gists. Python is a popular programming language and ranks among the fastest growing languages today. It follows only Ruby and Javascript in proportion of files in public gists [Wan15]. Python is frequently used for teaching introductory programming classes as well as used by non-professional programmers, such as scientists.

Previous research by Yang et al. [Yan16] examined Python snippets on Stack Overflow and found that only 25% were runnable (but did not investigate why). In this work, we focus on examining gists shared on GitHub instead of Stack Overflow. As observed by Sillito et al. [Sil12] and Yang et al. [Yan16], code snippets on Stack Overflow are often mixed with exposition and code, making it difficult to understand which segments of code are meant to be executed in an automated analysis. Therefore, there is strong motivation to investigate the underlying reasons why Python code may not be executable and understand the effort involved in configuring environments capable of running it. These barriers can cause problems for learners and non-professionals programmers lacking system configuration skills.

2.2.2 Mining Gists

We consider two strategies for mining gists from GitHub. GitHub provides a REST API for public gists, however, there are several limitations. Currently, the API provides no support for filtering queries based on language type. Furthermore, the API limits requests to 3000 gists when using pagination. To overcome these limitations, it is possible to filter gists based on creation date, meaning that all gists could be slowly enumerated by strategically modifying the creation date as a filter.

Another strategy is to scrape gists from the GitHub gist search UI. The search UI allows several filters, such as star rating, language, and keywords contained in the gist. The UI returns at most 100 pages of 10 random gists matching a search, allowing 1000 gists to be returned per search. By strategically modifying search terms, it is possible to quickly discover gists that meet the desired criteria.

For our initial population of the Gistable database, we focused on the scraping approach, which allowed us to focus on a particular language and to better control the quality of gists while using less computational resources.

2.2.3 Environment Inference Algorithm

To perform environment inference, we use an approach which builds an Abstract Syntax Tree (AST) of the gist source code and extracts all declared imports. Extracted imports are then filtered to remove all packages which are part of the Python standard library. Imports are assumed to be part of the standard library if they are present in a Docker image containing a clean install of the Python runtime.

We use the assumption that each import represents a single package that needs to be installed, and that the import name matches the name used to install the package. This is not always the case. For example, the Python package `beautifulsoup4` is imported as `bs4`. However, developer practices from Section 2.3.3.3 suggest that this is a useful approximation because it is the natural first step a developer takes when attempting to configure a computing environment. Errors from packages which could not be found are ignored. Such packages are simply not included in the final environment configuration. This allows us to recover from potential errors in our inference algorithm.

2.2.4 Execution Harness

To deal with the large number of gists analyzed as part of the Gistable database, we built an execution harness on a distributed cluster using the HashiCorp Nomad job scheduler, which natively supports docker containers. The harness is responsible for running all gists through the validation process to first determine if environment inference is needed and categorize the result of gist execution.

To isolate effects of dependencies and other system wide configurations, we perform analysis

inside independent Docker containers. The container filesystem also guarantees consistent starting environments.

2.2.5 Using Gistable

Gistable provides a command line tool for interacting with gists from the Gistable database. Gists can be cloned into a specified directory using the command `gistable clone <id> [location]`. Behavior is similar to that of `git clone`, and gists are checked out to the working directory if no location is specified.

If Docker is installed and running on the system, the CLI can also be used to directly execute a gist and display all execution results. Just call `gistable run <id>`.

2.3 Methodology

2.3.1 Research Questions

In this study, we investigate the following research questions and offer the motivation for each:

RQ1 – Can gists be executed? Can the average Python gist on GitHub be run to completion, or will it raise an exception? If gists can be run to completion, then they already form a database of snippets that can be used in research. However, if, like the Python snippets from [Yan17], gists cannot be executed by default due to syntax errors or other runtime exceptions, then additional investigation is needed.

RQ2 – Can a naive algorithm enable exectuable gists? Can we apply a simple approach for resolving unmet Python dependencies to address most runtime exceptions? If a majority of errors can be addressed by a simple resolution strategy, then there are a limited number of cases where automated environment configuration is needed. However, if a simple approach cannot be used, then more research is needed for developing a more comprehensive automatic environment configuration technique.

RQ3 – Why might gists not be executable? If gists cannot be executed even after resolving package dependencies, the natural question is why. Are they missing configuration for environment variables, services, or other kinds of dependencies? Categorizing gist execution failure and finding common root causes may lead to insight into how to improve future automatic environment configuration techniques.

2.3.2 Data Collection

To address our research questions, we first focused on building a large dataset of Python gists. We used the mining procedure outlined in Section 2.2.2 to mine 10,259 Python gists. We limited our search criteria to gists with at least one star [Kal14]. Currently, GitHub contains 32,233 Python gists with at least star—meaning our sample represents nearly 31% of all public starred Python gists.

Figure 2.1 illustrates an example of a gist in our experimental dataset and its accompanying automatically created Dockerfile. The gist uses the Google Maps geocode api to retrieve the latitude and longtitude coordinates of Zurich, Switzerland. The Dockerfile bases the image off of a Python environment, adds the gist code file, installs `requests`, and configures the default command to run the gist. Note that the package, `json`, does not need to be installed as it is a default system package.

2.3.3 Analysis

To answer our research questions, we used the following procedures to analyze our data. The inference harness described in Section 2.2.4 was used to clone gists from GitHub and perform analysis. Using two `ubuntu-16-04-x64` worker nodes sized at `2gb` and running in Digital-Ocean, inference took approximately eight hours to schedule and run all jobs.

2.3.3.1 RQ1

To answer RQ1, we start by performing a baseline analysis of gists by attempting to execute them in isolated Docker containers based on the `python:2.7.13` and `python:3.6.5` images. Any gist which executed without error is considered to have exited with the code `Success`. Any non successful gist is coded by the name of the error which was raised. I.E., `SyntaxError`, `ImportError`, `NameError`, etc.

2.3.3.2 RQ2

Research from Becker et al. [Bec18] indicates that the practical approach when there are multiple failures is to focus on the first error until it is resolved, then move on. This follows from the observation that first failures are useful because they are informative, need to be fixed, and their resolution may reveal deeper errors that were not apparent before.

To answer RQ2, we focus on gists where the first encountered failure was an `ImportError` and ask if we can configure the environment with all necessary dependencies. A naive attempt is made at performing environment inference by applying the inference procedure described in Section 2.2.3. We attempt to install each inferred package with the Python package manager `pip`. This is based on our findings from Section 2.3.3.3, which showed that attempting to install a resource name listed in an import error is often the first step developers take when attempting to fix environment configurations.

After applying our inference algorithm, the gist is then executed a second time with the new environment specification, and the evaluation results recorded under the same criteria as for the baseline.

Table 2.1 Gists per exit code in the baseline evaluation using Python 2.7.13.

Result	Count	Percent
ImportError	5379	52.4%
Success	2501	24.4%
NameError	852	8.3%
SyntaxError	753	7.3%
IOError	167	1.6%
IndentationError	153	1.5%
SystemExit	115	1.1%
EOFError	94	0.9%
OSError	48	0.5%
ValueError	34	0.3%

2.3.3.3 RQ3

We performed a random sampling on failing gists in order to understand why they failed to execute. For this analysis, we performed descriptive coding [Sal09] and composed *memos* [Bir08], which described several reasons for a gist failing to execute. These memos captured interesting events or properties of environments and code snippets to promote depth and credibility, and to frame the information needs of an automated environment configuration technique. That is, they provide a *thick description* to contextualize the findings [Pon06].

We then solicited 24 developers familiar with Docker to manually inspect gists. Each developer was given a disjoint random set of 10 gists and asked to create a Dockerfile that would enable successful execution of the snippet within a standard time period (one and a half weeks). The developers had between 6 months to 5 years of industry experience and familiarity with Python. Further, the developers had been trained in several workshops on configuration management skills, including Ansible and Docker.

We asked the developers to rate the difficulty of creating a Dockerfile and the steps they took to create it. We then performed a qualitative coding exercise over the Dockerfiles and reported steps using closed codes derived from our first qualitative coding. During the coding process, we employed the technique of *negotiated agreement* as a means to address the reliability of coding [Cam13]. Using this technique, the first and second authors collaboratively code to achieve agreement and to clarify the definitions of the codes; thus, measures such as inter-rater agreement are not applicable.

2.4 Executability Results

2.4.1 RQ1 – Can gists be executed?

Table 2.1 provides the names and counts for the most common reasons a gist terminated when run in an isolated Python `v2.7.13` environment.

Consistent with the Yang et al. [Yan16] study on Stack Overflow Python snippets, we

observed that only 24.4% of Python gists were executable. The majority of gists (52.4%) failed to execute due to an `ImportError`, which is typically caused when a python dependency could not resolved or loaded. We observed that only 17.1% of gists failed to parse (i.e., `SyntaxError`, `NameError`, and `IndentationError`. Our observed rate of parse failures for gists is slightly lower when compared with Yang et al.'s observed rate of 25% for Stack Overflow snippets. We believe this may be caused by the difficulty of distinguishing exposition from code when parsing code snippets found on Stack Overflow [Sil12]. For example, in a Stack Overflow post, it could be common to include code and output typed into an interactive shell in order to help explain a concept, which is not directly parsable.

Finally, we observed <8% of gists failed to execute due to some other runtime exceptions, such as `IOError` or `OSError`. These failures could be caused by missing resources, such as files, services, or platform specific dependencies.

Baseline results for executing in a Python 3 environment show 3,907 instances of `SyntaxError`, compared to the 753 for Python 2. In addition, the number of gists which exited with `Success` dropped to 1,445. The number of gists which exited with `ModuleNotFoundError`, a direct subclass of `ImportError` in Python 3.6, was 3,353. While this shows a decrease from the 5,379 in Python 2, the large set of `SyntaxError` may shadow an undetermined set of gists which would also see an `ImportError`.

> Overall, we find that most gists are not executable in a default Python environment. Further, the exceptions raised when attempting to execute the gists suggests that an insufficiently configured environment is the primary cause.

2.4.2 RQ2 – Can a naive algorithm enable exectuable gists?

The baseline analysis for RQ1 showed the majority of Python gists require environment configuration. To determine if a simple algorithm is capable of resolving such errors, we applied our inference algorithm described in Section 2.2.3 to the 5,379 gists which failed due to `ImportError` using Python 2, attempting to install all third party imports with `pip` in both a Python 2.7.13 and Python 3.6.5 environment. Python 2 is used as a baseline for `ImportError` due to its lower frequency of `SyntaxError`.

We analyzed each gist after attempting to install all inferred dependencies and recorded the exit status according to the same criteria used for answering RQ2. For Python 2, 2,488 gists exited due to a reason other than `ImportError`, a gain of approximately 46%. Of these gists, 1,294 finished with `Success`. The remaining 1,194 finished with some error other than `ImportError`. When also considering Python 3, the number of gists which had become executable increased to 2,870. Overall, considering Python 3 resulted in an additional 428 gists becoming executable after inference when compared with only using Python 2.

While a naive approach can infer dependencies for some gists, it fails to do so in the majority case.

2.5 Execution Failures

To answer, *RQ3 – Why might gists not be executable?*, we inspected the gists to better understand why they failed to execute, even after applying our naive algorithm. First, we focused on gists failing with `ImportError`, which was the most common failure status. Then, we also inspected gists which failed for other reasons, such as `IOError`. Finally, we characterize the effort reported by developers when manually creating Dockerfiles for the failing gists.

2.5.1 Gists Failing with ImportError

We report our findings in Table 2.2. Overall, the 24 developers participating in this study were able to submit a response for 218 out of the 240 gists assigned to them as a group. The average number of Dockerfiles received from each developer was 9, with a minimum of 3 and a maximum of 10.

In addition to the failures reported in Table 2.2, 24 gists were considered flaky. Inference of flaky gists may have failed due to network or memory issues. One developer reported needing to increase the memory Docker was configured to use in order to properly install dependencies for one such gist.

Collectively, the developers indicated that they were unsuccessful in creating a working Dockerfile for an additional 78 gists. The feedback we gathered for such gists showed that even developers familiar with environment configuration may be unable to correctly deduce the correct specification for an arbitrary snippet of code. One developer, after referring to an existing Dockerfile related to the gist they were working with, wrote

> *I attempted to adapt the Dockerfile listed above to run this gist, but was never able to get it working; needless to say I would not have been able to do it without the Dockerfile listed either; I attempted various other ways to install the android sdk (apt-get, etc), all of which failed; constantly ran into 404 errors with apt-add-repository; got "No space left on device" error when running listed Dockerfile in a virtual machine; the Dockerfile built when running natively, but I could not find a way to use the "monkeyrunner" command, as this gist is supposed to be run with "monkeyrunner" and not "python" (from what I understand); a great deal of time spent trying futilely to get this to work.*

We now focus on a selection of distinct failure causes.

Names. The most common case, as stated in Section 2.4.2, is when a resource name does not match the name of the package it belongs to. Resolving this situation often required the

Table 2.2 We had 24 developers familiar with environment configuration techniques attempt to manually create Dockerfiles for 218 of the gists for which naive inference failed to resolve import errors. This table summarizes reasons for failure as reported by the developers, focusing on the first failure reported. We manually inspected each gist in cases where no clear reason was found by a developer, applying our own failure category if possible, or labeling the gist as unconfirmed.

Cause	Count	Example
Package name did not match the resource imported in the gist	70	`https://gist.github.com/syl20bnr/6623972`
Gist dependencies have additional dependencies which need to be resolved	23	`https://gist.github.com/kennethreitz/2901479`
Relies on missing C library files or headers	16	`https://gist.github.com/huyx/8069261`
Requires a previous version of a package due to breaking changes	15	`https://gist.github.com/segphault/9f2d7da68779a17a0890`
Dependency can only be installed on a non-linux operating system	13	`https://gist.github.com/mapleray/4189391`
Relies on a standard package that was introduced in a later version	12	`https://gist.github.com/fmasanori/4684752`
Pip errored during installation, possibly timing out on large packages or propagating an exception raised by the package	12	`https://gist.github.com/willwade/5330566`
Unconfirmed. The exact failure could not be narrowed down to a single category.	9	
Gist is missing necessary environment configuration, such as settings files	8	`https://gist.github.com/Sinkler/bfc2099235ac96937f34`
Dependency wasn't available on PyPI, nor installable via the Ubuntu aptitude package manager.	7	`https://gist.github.com/JudoWill/764262`
Dependency is only supplied as part of a custom execution environment or interpreter	6	`https://gist.github.com/Utopiah/a2b9c6ecdb24ca8fd6f4f41a9c0eb32e`
Relies on a deprecated package that is no longer maintained and is no longer available to be installed	1	`https://gist.github.com/matbor/6532185`
Gist is not intended to be run and imports libraries which don't exist	1	`https://gist.github.com/RichardBronosky/454964087739a449da04`
No versions are available for install	1	`https://gist.github.com/mclavan/276a2b26cab5bc22d882`

developer to search the package index, test multiple packages, or query developer resources such as Stack Overflow.

For example, one gist relied on the module named `i3`, but the developer found they had to install the package `i3-py`, resulting in the following Dockerfile:

```
FROM python:2.7.13
ADD i3_focus_win.py /
RUN pip install i3-py argparse
CMD ["python", "/i3_focus_win.py"]
```

System dependencies. Missing C libraries were also a common issue. Many Python dependencies serve as bindings into C libraries installed as a system dependency, and fail to compile on installation because the system dependency is not present. In some cases, a dependency failed to compile because the Python Docker image did not include C build tools, such as cmake, that they relied on.

One such gist made use of the Python `hunspell` package, a wrapper for the C program Hunspell. The developer found that before using pip to install `hunspell`, they needed to add `RUN apt-get install libhunspell-dev -y` to their Dockerfile.

Custom environments. In some cases, a dependency was distributed as part of a separate execution environment. For example, one developer reported that a gist relied on the `bpy` module that ships with Blender. After installing Blender and still seeing an `ImportError`, the developer discovered a Stack Overflow post saying `bpy` can only be imported when running in Blender's bundled Python interpreter.

Unlisted packages. Several gists depended on packages which were not available through the PyPI or Aptitude package managers by default. Such packages require being installed from a separate repo, such as an Aptitude Personal Package Archive (PPA) or directly from a git based public repo.

In one example, a user commented on the Gist that they had difficulty importing one of the modules, even though they had installed the correct package.

> *ScissorPush?*
>
> *from kivy.graphics import ScissorPush ImportError: cannot import name ScissorPush*

Resolving this issue required installing an unreleased version of python-kivy that needed to be installed from a PPA.

```
FROM ubuntu:16.04
RUN apt-get update
RUN apt-get install -y software-properties-common python-software-properties
RUN add-apt-repository ppa:kivy-team/kivy-daily
RUN apt-get update
RUN apt-get install -y python-kivy
ADD snippet.py /snippet.py
CMD ["python", "/snippet.py"]
```

Deprecated packages. In other cases, gists relied on packages that are no longer maintained and can no longer be installed. Common causes are not supporting SSL, which pip now requires, not fixing known bugs which prevent installation, or even an entire package no longer being provided for distribution.

For example, the Python Quartz package has an omission in the manifest that prevents the requirements file from shipping with the package source. The developer is aware of the issue, but has declared they will not create a patch.

> *To fix this problem, I have to include requirements.txt in MANIFEST.in so that the file will be shipped with the sources.*
>
> *Unfortunately, I abandoned this project a while ago and I am currently working on a complete rewrite...*

Sometimes, a package is still actively maintained, but the gist relies on features from a version which had reached end-of-life and is no longer being distributed.

Configuration settings. Some gists require additional configuration files which are not provided with the gist itself. For example, it was common to read in secret keys and values from a non-existing app.config file in order to read a setting such as `TWITTER_API_KEY`. These configuration files are not preexisting dependencies which can be installed.

Language version. Python 3 has introduced several new modules, like `urllib.request`, that are not present in Python 2. Gists that rely on these modules must be run in a Python 3 environment, and are incompatible with the `python:2.7.13` Docker image being used. In some cases it may still be challenging to determine which Python version to use. For example, `pathlib` is a part of the Python 3 standard library, but was not introduced until Python 3.4, and support for it was only added to the standard library in Python 3.6.

Operating System. Developers also saw dependencies which could only be installed on a specific operating system, such as Windows or macOS. One developer, when asked to create a configuration for a gist, found that the gist was designed to interact with the Windows registry, and reported

> *Packages are dependent on Windows (not Ubuntu).*

Such gists cannot be run in the Ubuntu based Python image.

2.5.2 Other Failing Gists

To characterize the gists in our dataset and gain a better understanding of how they are used on GitHub, we computed basic metrics across all gists using tools developed for our execution harness. Additionally, we performed an inspection on 30 randomly selected gists from the 10.3k in our dataset with the focus on characterizing what resources they might rely on, including, but not limited to, dependencies.

Our random sample found that 14 out of 30 gists (46%) did not rely on a third party package. Approximately 13% did not import any packages, and 76.7% relied on Python library packages. 6.7% optionally loaded a third party package if it existed in the environment. We found that many gists rely on connecting to networked resources, or on interacting with configuration files and executables on the file system. Other gists required interaction from the user in some manner, either requiring input over `stdin`, command line arguments, creating an interactive prompt, or displaying information through a graphics interface. In the worst case, a gist does nothing because it is either recognizably not correct Python syntax, or because it defined classes or functions but did not otherwise execute code. This happened nearly 10% of the time.

Overall, the gists in our dataset import over 1,700 unique third party packages and on average have 92 lines of code.

2.5.3 Developer Extraction Effort and Effectiveness

The median difficulty rating reported for configuring a gist was 3 on a scale of 1-5, reported for 24.3% of all gists. Only 13.7% of the gists were reported as very easy to execute by our developers, whereas 22.4% were reported as very difficult to execute. Developers reported spending between 20 minutes to 2 hours to setup the environment for executing each gist.

Of the 140 gists developers found an environment configuration for, the average Dockerfile was less than 10 lines and installed less than 5 packages. However, we found that not all of the submitted Dockerfiles were capable of executing their gists without `ImportError`. For example, one developer submitted the following Dockerfile, claiming that the gist ran without any errors in the provided environment.

```
1 FROM python:2.7.13
2 ADD https://gist.githubusercontent.com/awesomebytes/cb5a28fa8d4db3fc1ba51894663c1aed/raw/
  ↪ cba597a5219d807c5e4940e9d2018d47b5eca809/watson_ros_publish_string
  ↪ /snippet8.py
3 RUN pip install ws4py
4 CMD ["python","/snippet8.py"]
```

However, we found that executing the gist still failed with the configuration error `ImportError: No module named rospy`. One interpretation is that not only can this be a time-consuming task for developers, but the process can be also error-prone.

2.5.4 Developer Responses

Section 2.5 revealed common properties of gists that made environment configuration difficult. We now highlight a selection of developer responses which illuminate the process that developers employ when faced with these challenges.

Version errors. Developers reported several experiences related to resolving errors that were present due to mismatches in versions of dependencies and code.

django was the only import required. But that didnt simply resolve the error. There was a import error for CompatCookie. Tried in python 3 as well but no luck. Later found out from django release notes that it was deprecated after v1.4. So tried to pip install older version of django and was finally able to resolve the import error. Docker file builds and runs without any error.

Another developer described how the requirements could shift depending on the version of a dependency used.

Spent over an hour to find the imports needed for text.blob. It was replaced to textblob from version 0.7.1 and when I tried the lower version I received another error that required dependencies on a higher version.

Unlisted or unknown dependencies. Developers reported several instances where they had difficulty determining the provenance of a dependency.

1. Git clone basic-python-logger repo from https://github.com/vehrka/basic-python-logger (Have to google and find out that basiclogger.py is not a module, but rather a script wrote by the creator of the Gist itself) 2. pip install psycopg2, pandas, sqlalchemy for satisfying the dependencies. 3. Upon doing this, the images builds successfully

Another developer had difficult working with a cloud provider package.

Couldn't find clouddns module. Couldn't solve dependency. Spent 2 hours on it.

Resource limitations. Several developers reported experiences related to memory or disk limitations on their personal computers when building environments.

MEMORY ERROR while installing keras and pyspark resolved by --no-cache-dir flag

2.5.5 Summary

We conclude RQ3 with the following observation:

> Python gists often require non-trivial environment configuration in order to run. There are multiple reasons why configuration for any particular gist might be difficult, but the most common challenges are finding dependencies without obvious names and installing dependencies with transitive dependence on system modules.

2.6 Discussion

2.6.1 Towards automated environment configuration and beyond

While the inference procedures presented in Gistable are simple, we show that they successfully lead to a correct environment specification in a number of cases. This indicates that reliable

environment inference is possible, and highlights areas of research where techniques can improve. For example, we may consider combining dynamic inspection of packages and machine learning algorithms for inferring possible environment specifications.

Although we focus on Python gists, we believe our insights can generalize to other programming languages. For example, dependency on system build tools can also be a problem in the Node.js ecosystem when packages compile native addons. Common compilation troubles in Node eventually prompted Microsoft to publish developer guidelines [1].

Given a language which supports third-party packages, our approach only assumes two things. First, that packages have a set of named resources that they make available for use by client code. Second, that the identifier for a package resource used by client code has at least a substring match for a resource provided by the package. This is the case for popular languages like Javascript and Ruby, and so we believe that our approach will generalize to these, and similar, languages.

Furthermore, our inference procedure only requires a code snippet, and could easily be modified to work with another context, such as code snippets in Stack Overflow answers, blog posts [Par13], or online documentation [Tre18]. Gistable focuses on configuring and running single file scripts. However, many projects have a large number of interacting tools that make configuration challenging. We believe insights from our work can inform configuration techniques for larger projects in the long-term.

2.6.2 Challenges in mining gists

Querying for unique gists isn't directly possible

Instead, we rely on manipulation of search parameters in the GitHub UI to return results. On subsequent runs, the gists returned by a UI search are often different, allowing the use of a dictionary approach for collecting unique gists by ID. However, some gists may still be duplicates, either by forking or simple duplication of content. Forked gists have metadata available indicating the origin, but, in the worst case, it is generally undecidable if two gists are equivalent.

Gists can be complex

While most of them are relatively simple, there is no requirement that a gist consist of only one file, or even of files in a single programming language. If a gist has more than one file, the entry point is often ambiguous, unless the programming language runtime supports running a default file, and such a file exists in the gist. We discarded gists with more than one file to avoid having to deal with this situation.

2.6.3 Challenges in automated configuration inference

There are several challenges identified by our work.

[1] https://github.com/Microsoft/nodejs-guidelines

Name resolution

An important task in automatically creating an environment specification from code is: *given a code snippet, infer the set of installable packages associated with the code*. Luckily, package import statements within the code snippet can help; however, there are still several complications that must be resolved. In the simplest case, many package names may not match the name they are imported by (e.g. the `i3`/`i3-py` mismatch encountered by one of the developers participating in the study).

Another consideration is that many gists have imports structured as follows: `import kazoo.client`. In our evaluation, the naive algorithm attempts to install `kazoo.client`, and fails. The actual package is `kazoo`. However, in other cases, like `zope.interface`, the appropriate package name is indeed `zope.interface`. Finally, it could be possible that some code snippets are incomplete; that is, they may omit import statements for packages being used in code.

A first step to addressing this challenge may be to preprocess known packages by extracting a list of resources that each exports. When performing inference, resources might be mapped to installable packages by a reverse look up. However, this introduces its own challenge of dealing with packages which have conflicting resource names.

System Dependencies

Other packages have implicit dependence on system environment configuration or other system packages. Unfortunately, this type of error often presents itself as a compile time error when a header file cannot be found. Header files, like package resources, do not necessarily have a name related to their project. Like the Python package name resolution challenge, this could be addressed by preprocessing and reverse look up. Another option is to analyze existing configuration scripts for Python projects and perform association rule mining to infer dependence between a Python package and a system dependency.

Language Version

Even with a gist consisting of a single snippet in the desired language, it is often non-trivial to decide which language version to use. For our Python gists, most of them are capable of running successfully with Python 2. However, reported instances of `SyntaxError` may be due to use of syntax created in Python 3. Gists can be checked for syntax errors by attempting to compile them, so Python 2 or Python 3 syntax compatibility could be checked by compiling under each language runtime. Python 3 dependence may also be inferred by checking gist imports against the Python 3 standard library.

Unlisted and Deprecated Packages

Packages may not be installable from a general package repository, like the Python package PyTorch. In this case, it may be possible to install directly from a git based repo, if one can be

inferred from previously seen configuration scripts.

2.6.4 Future Applications

While the focus of our paper was evaluating the executability of Python gists, we envision several additional research applications for Gistable.

- **Text summarization of tasks:** Recent work ([Luc14; Hai10; McB14; Edd13]) has focused on performing semantic code summarization. Because gists typically correspond to idiomatic programming uses and tasks, there is an opportunity to use gists as a dataset for learning models which support semantic summarization of code.
- **API usage analysis:** We observed over 1,700 unique third-party python packages in our initial version of Gistable. This suggests that gists can provide a rich source of information for mining and understanding how APIs are used in practice by programmers.
- **Test input generation:** Gists often have hard-coded input text for running the code example. An interesting research opportunity would be to use gists as a benchmark for generating test inputs that can also successfully run (or fail) in a gists.
- **Configuration repair:** In addition to inference of configuration environments, it is possible to support research in repair of configuration scripts [Wei17]. For example, if a user updates the code to use new library, pushes changes to production, but did not update the Dockerfile [Sch18], an exception can be thrown. However, a configuration repair tool can suggest a repair that updates the environment specification to use the correct version of the package: e.g., `"networkx==2.0"`, which eliminates the exception.
- **Resource repair:** Gists may have external resources, such as URLs, or publicly hosted APIs, such as the Google Maps API. One interesting application would be to study the decay of resources overtime (bitrot). Further, if resources or API URLs change, is it possible to repair the invalid resources and code?
- **Understanding the Python community's use of gists:** Wang et al. [Wan15] studied the use of Public gists on GitHub, observing a variety of uses, but did not focus on usage categories per language or file type. We observed several different usage patterns for Python gists in the Gistable database. Future research can inspect and categorize the types of practices that emerge from creating and sharing public gists in the Python community.

2.7 Limitations

Our analysis may overestimate the number of executable snippets. For example, a gist may define only a single function containing all code, but never call it. So long as the function definition succeeds, the gist is marked as successful regardless of whether or not the code works. Any

additional dependency errors caused by executing the function will not be triggered. Further, our analysis may misclassify an exception. For example, it is possible for a gist to implicitly hide import errors by catching them and then raising an error of a different type. Future analysis can use several measures to increase the certainty of successfully execution: annotating gists with test assertions, increasing path coverage of executed gists, manual inspection and verification, and iteratively fixing configuration issues and evaluating gist execution.

Another concern is that running a gist once will only produce the first fatal error encountered, although the gist may have more than one. As a result, we may underestimate the distribution of some category of errors. However, research from Becker et al. [Bec18] argues that the practical approach in such a case is to focus on the first failure encountered, as this mirrors how developers typically resolve errors.

We caution readers to not overgeneralize our results. While we analyzed a large sample of Python gists, these results may not extend to other programming languages. Numerous factors, such as the experience of programmers, the quality of modules and package management, the degree of third-party modules usage, and language design can influence how executable a code snippet is in practice. In other languages and ecosystems, these factors may be less of a concern. Further, we examine public gists, which may differ from private gists.

Our environment inference algorithm can have several limitations. Even if all dependencies install correctly and gist execution succeeds, there is no guarantee the package API will not undergo a breaking change between the time the Dockerfile is created and the time the image is built.

2.8 Related Work

The work by Yang et al. [Yan16] is the closest related work in terms of research approach and methodology. Yang et al. [Yan16] examined Python snippets on Stack Overflow and found that 75% were parsable and only 25% were runnable. In this paper, our work differs in several important ways. First, our primary focus is on the ability to execute Python snippets, whereas Yang et al. were primarily focused on the ability to parse snippets. Second, we investigate the effectiveness of a naive inference algorithm in recovering an execution environment for the snippets. Third, we manually construct execution environments and characterize reasons why code may not be executable. Finally, our research context differs in that we examine gists shared on GitHub instead of code snippets found in Stack Overflow answers. Overall, our research complements Yang et al.'s [Yan16] work in understanding challenges for sharing and using snippets on the web, while providing new directions for research in automated configuration inference.

Several researchers have characterized the buildability of software projects. Sulír and Porubän [Sul16] performed a study on 7,200 Java projects and studied the ability to automatically build them by attempting a maven or ant build in a virtual machine. They found that more than 38% of builds

ended in failure. The authors identified that the largest portion of errors are dependency-related. Incidentally, Urli et al.'s study [Url18] on program repair of Java programs is related to our work. Urli et al. found that by attempting to automatically build 1,609 Java projects on GitHub with Maven, they could only reliably reproduce 31.82% of test failures due to the complexity of mimicking configuration for test environments. A notable difference is that our work focuses on automated configuration inference, whereas Urli et al. focus on repairing Java code in order to pass test failures and thus does not investigate why projects could not build or run tests. *Buildability* and *executability* are related yet distinct concerns in software maintenance. First, build failures can be associated with difficulty inherent in build maintenance that is independent of reproduction. For instance, McIntosh et al. [McI11] find that the effort involved in maintaining the build configuration can introduce 27% overhead on source code development and a 44% overhead on test development. Such high effort could increase the odds of out-of-date or non-buildable projects. Second, while building large and complex projects can be daunting, this process does not necessarily *run* the code, which can require further environment specifications. Finally, our research context differs from buildability of software projects in that we are interested in automatically executing isolated code snippets without build specifications, which is common in learning or documentation contexts.

German et al. [Ger07] describe multiple problems associated with managing and specifying dependencies, including downloading, building, and satisfying inter-dependent artifacts, which may not always be explicitly documented. They propose a framework for categorizing dependency types and a method for building and visualizing an inter-dependency graph of a package. Lungu and colleagues [Lun10] note that dependencies also exist between projects in a software ecosystem. They propose a model which can capture inter-project dependencies. In our work, we are interested in characterizing both dependencies as well as other environment resources that when absent can prevent code from being executable. We believe our empirical findings complement these models and together, they can be used to inform the design of an automated configuration inference tool.

2.9 Conclusion

Code snippets can be a useful way to explain and demonstrate a programming concept, but may not always be directly executable. We investigated the executability of Python gists hosted on GitHub and the ability for a naive inference algorithm to recover a Dockerfile capable of executing the Python gist. Finally, we investigated the types of execution failures encountered when running Python gists and the effort involved in manually creating a Dockerfile able to run a gist.

Overall, we find that most gists are not executable in a default Python environment. Further, the exceptions raised when attempting to execute the gists suggests that an insufficiently configured environment is the primary cause.

Our inference algorithm shows that, at least in some cases, correct application environment configurations can be automatically recovered. While a naive approach can infer dependencies for some gists, it fails to do so in the majority case. Additional strategies promise greater success, and will be the subject of future research.

Our investigation of Python gists finds that they often require non-trivial environment configuration in order to run. There are multiple reasons why configuration for any particular gist might be difficult, but the most common challenges are finding dependencies without obvious names and installing dependencies with transitive dependence on system modules.

Finally, we envision multiple applications for Gistable that extend beyond empirical studies of executability. Gistable can automatically configure and execute approximately 5,000 public Python gists hosted on GitHub. Each gist has an accompanying Dockerfile which can be used to build a Docker image based off of the `python:2.7.13` image which contains both the gist and its dependencies. Running the Docker image executes the gist without `ImportError`. Gistable also ships with a simple command line utility for cloning gists in the dataset, and building and running Docker images.

CHAPTER

3

INFERRING APPLICATION DEPENDENCIES

This work originally published as Horton, E. & Parnin, C. "DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets". *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 328–338

Sharing code snippets to illustrate a specific task is frequently used practice within the software engineering industry [Sil12]. Due to the importance of sharing examples, platforms like Stack Overflow and GitHub's gist system have been created to facilitate social learning [Par12; Yan16; For16] through community driven interaction. GitHub gists are short but complete programs, often only a single file.

Unfortunately, many code snippets do not contain information regarding the system configuration needed for proper execution, as system configuration is not an inherent property of code. Consider Sentry, an error reporting system. The official client for Python, Raven, supports the Flask framework [Sen]. Examples[1] demonstrating how to use Sentry with Flask often have no indication that it needs to be installed with Flask extras, causing developers to encounter runtime errors.[2]

[1] `https://gist.github.com/1cdd9646046ae10c2932`

[2] `https://github.com/getsentry/raven-python/issues/1075`

This is a wide-spread problem. Research by Yang et al. found that only 25% of code snippets from Stack Overflow can be run without error [Yan16]. Horton and Parnin later found 24.4% of Python gists from GitHub to run without error [Hor18]. The main cause of failure, experienced by 52.4% of the gists evaluated, was a dependency error. Seo et al. also found that approximately 50% of build errors are caused by dependencies [Seo14]. Further, the effort involved in manually constructing an environment specification is non-trivial — developers can spend between 20 minutes and 2 hours creating a Dockerfile for a single code snippet, and often fail to construct a valid specification [Hor18]. Common challenges include mapping a code resource to its originating package and determining the correct order of installation for transitive dependencies.

This work focuses on automating the dependency resolution process of system configuration management for both language-level and system-level dependencies. Before performing dependency resolution, we build an offline knowledge base from two sources. First, we process existing packages on the Python Package Index (PyPI) by extracting declared resources and using dynamic analysis to determine possible dependencies. Second, we inspect project configuration files from GitHub and generate association rules for pairs of packages which are frequently seen together. Environment inference for a code snippet is performed by querying the knowledge base to map code resources to installable packages. A search algorithm then resolves all transitive dependencies in a consistent order.

We implement DockerizeMe, a tool for applying dependency resolution to a code snippet and generating a Dockerfile for the corresponding environment. Unlike other approaches to automated software configuration, which focus on repairing configuration errors ([Wei17; Has18; Ren18]), DockerizeMe focuses on inferring a complete configuration without external inputs. Being able to automatically infer code dependencies has the potential to save developers time, reduce costs of learning and development, and enable repair and verification of code snippets in online platforms. It is also a first step towards fully automated software configuration management.

To evaluate DockerizeMe, we performed environment configuration for 2,891 gists from the Gistable dataset which still failed due to Python's `ImportError` after applying Gistable's Environment Inference Algorithm (Section III-C from [Hor18]). DockerizeMe successfully removed import errors for 892 gists.

3.1 Motivating Example

Many code snippets shared as examples are not directly executable, often because they depend on external libraries that are not present by default on a developer's system [Yan16; Hor18]. Discovering all required dependencies is a time consuming process that even experienced developers have trouble with [Hor18]. Consider the following scenario:

A developer is working on a networking component and needs to parse packets from a network interface. They would like to use Python, and search for Python libraries that can be used to perform this task. Fortunately, other developers have previously asked for recommendations on

```python
import pcapy
from impacket.ImpactDecoder import *

# list all the network devices
pcapy.findalldevs()

max_bytes = 1024
promiscuous = False
read_timeout = 100 # in milliseconds
pc = pcapy.open_live("eth0", max_bytes, promiscuous, read_timeout)

pc.setfilter('tcp')

# callback for received packets
def recv_pkts(hdr, data):
    packet = EthDecoder().decode(data)
    print packet

packet_limit = -1 # infinite
pc.loop(packet_limit, recv_pkts) # capture packets
```

(a) snippet.py, a code snippet showing how to capture packets on a network interface.

```dockerfile
FROM python:2.7.14

# Update APT package list and install required system dependency
RUN apt-get update
RUN apt-get install -y libpcap-dev

# Install Python dependencies not included with snippet.py
RUN pip install pcapy
RUN pip install impacket

COPY snippet.py /scripts/snippet.py
CMD python /scripts/snippet.py
```

(b) Dockerfile for running the code snippet.

Figure 3.1

libraries to solve the same problem. They quickly come across a post on Stack Overflow[3] with a couple different recommendations.

The accepted answer recommends Scapy, but it is licenced under GPLv2, a copyleft license which the developer cannot use for their project. However, another answer recommends Pcapy, and provides an example snippet for printing packets as they arrive on an interface. The developer wants to see if the example works, so they create a file named snippet.py (Figure 3.1a) containing the example code, modifying the network device name in the example, "name of network device to capture from," to be "eth0." However, when running `python snippet.py`, they are met with the error `ImportError: No module named pcapy` due to the fact that pcapy is not a part of

[3]https://stackoverflow.com/questions/4948043/

the Python standard library.

The developer notes that there are two packages imported in the snippet, Pcapy and Impacket. Both packages exist on the Python Package Index (PyPI), so the developer attempts to install each with `pip install pcapy impacket`. Unfortunately, Pcapy fails to install due to a compiler error. Further investigation reveals that Pcapy relies on the pcap system library headers that the developer does not have installed. The developer first attempts to install the package pcap using apt-get, their system's package manager, but no such package exists. The actual package name is libpcap0.8. However, the pcap library does not come with the headers that Pcapy requires, and the developer finds they must install the development package libpcap-dev. The final configuration is encoded by the Dockerfile in Figure 3.1b. Without any aid, developers face a trial-and-error struggle to discover dependencies for environment specifications such as this.

3.2 DockerizeMe

The main purpose of DockerizeMe is to solve the dependency resolution problem in software configuration management. We now define the dependency resolution problem: given an runnable code snippet C, correctly install all *language-level* and *system-level* software packages required for C to execute without an import error. *Language-level* dependencies are dependencies managed by a package manager or tooling provided with the language runtime environment. A *system-level* or *system* dependency is installed on the system, but managed externally to the language runtime environment.

In the context of dependency resolution, a code snippet C is considered *runnable* if it can be evaluated by the execution environment. That is, it does not experience fatal errors at compile or load time. A *runnable* code snippet may experience a fatal error during runtime. We say C experiences an *import error* if it experiences a fatal runtime error caused by the failure to find a requested library.

We focus on Python, a popular language with a robust ecosystem containing over 146,000 packages on its standard package platform [Pyp]. A Python code snippet experiences an import error if it exits due to Python's exception `ImportError`. We begin addressing dependency resolution by building an offline knowledge base (Sections 3.3 and 3.4). We then design an inference algorithm (Section 3.5) to return dependencies in a feasible installation order.

DockerizeMe[4] is implemented as a NodeJS command line utility. Running `dockerizeme` on a Python package will generate the contents of a proposed Dockerfile containing all dependencies recovered by the inference procedure.

[4] `https://github.com/dockerizeme/dockerizeme`

3.3 Knowledge Acquisition

DockerizeMe uses an offline knowledge base to correctly infer dependencies for a target script. This knowledge base contains packages, their versions and resources, and the relationships between them. It is built by applying static and dynamic analysis to known packages from the Libraries.io [Tid18] dataset. Static analysis enumerates packages' known resources for later retrieval, and dynamic analysis gathers information about transitive dependencies. Association rule mining of dependencies in public Python projects takes advantage of developer generated knowledge of system level transitive dependencies. We now discuss each technique in detail.

3.3.1 Discovering Package Resources

Inferring which packages correspond to code resources used by a script can be a challenging and non-trivial task. As reported by [Hor18], many resources have a different name than the package that they belong to. It is often difficult for developers to determine which packages to use.

To better inform our inference procedure, we analyzed the top ten thousand Python packages on PyPI based on their SourceRank in the Libraries.io dataset [Hej18]. If the install was successful, we recorded the distribution's top level resources as listed in `top_level.txt`. For example, we extracted the resources Bio and BioSQL from the Python package biopython. Installation succeeded for 88% of the tested packages.

Some packages may have failed to install due to missing dependencies or some other unknown configuration. When this happened, we attempted to download and parse the package distributions manually. All packages were downloaded with pip using the options `--no-cache-dir` and `--no-deps`. If the package provided a wheel (a distribution in Python's binary format) on PyPI, we downloaded it with `--only-binary=:all:`. If the package did not have a wheel on PyPI, but did have a source distribution, we downloaded it with `--no-binary=:all:`. For source distributions, we then attempted to build a wheel distribution using the option `--no-deps`. If successful in either downloading or building a wheel for a package, we then parsed the package's top level resources by finding and reading the wheel's `top_level.txt` file. This was successful for one in three packages.

3.3.2 Dynamic Analysis

Some packages may not properly list their dependencies, preventing `pip` from automatically handling resolution during install. We address this issue by performing dynamic analysis, using the 10,000 packages by SourceRank from the Libraries.io data. First, we attempt to install each package using `pip install <package>`. If the installation succeeds, we then parse the top level resources and attempt to import each. Any error output from the install/import process is logged, and on failure we parse the output for instances of the following patterns, which indicate dependence on some Python package that was not present.

- `no module named <name>`.
- `pip install <name>`.
- `cannot find <name>`.
- `cannot import name <name>`.

For example, attempting to install the Python package PyHum ([Pyh]) results in the following output:

> *ImportError: No module named numpy. Please install numpy first, it is needed before installing PyHum.*

Based on the output, our dynamic analysis procedure enters a dependency record into the knowledge base which indicates that PyHum requires numpy.

3.3.3 Association Rules

Static and dynamic analysis cannot provide meaningful information about a package if the installation fails and no wheel can be found or built. Dynamic analysis may also fail to find a package's dependencies due to non-standard error messages or references to unknown header files in C libraries. In other cases, dependencies may be optional, or only required in conjunction with another package. This is the case for a simple Flask app using Raven Sentry for error logging.

```
1  from flask import Flask
2  from raven.contrib.flask import Sentry
3
4  app = Flask(__name__)
5  sentry = Sentry(app)
```

Running this Flask app, after installing Flask and Raven, will result in `ImportError: No module named blinker`. The system must also have blinker, an object signaling library, installed for Raven to correctly communicate with Flask.

DockerizeMe addresses these issues by augmenting its knowledge base with rules learned from existing Python environment configurations. We target public GitHub repos with a Dockerfile containing install commands for both apt and pip. The list of target repos was mined from Google BigQuery.

3.3.3.1 Extracting items from Dockerfiles

We inspect each repo's Dockerfile to find all `RUN` commands in both the exec and shell formats. Commands are normalized to remove new lines and escape characters. If the command is in exec format, it is additionally converted into a single command string. If a command string contains more than one command separated by `&&`, `||`, or `;`, it is split into a list of individual commands.

All tokens starting with "-" are assumed to be command flags and are ignored. Of the remaining tokens, commands which start with `apt-get install` are parsed for apt packages, whereas commands starting with `pip install` are parsed for pip packages. Additionally, parsed apt packages are validated to exist by checking against a list of known apt packages using the apt-cache utility. Parsed pip packages are validated to exist by checking PyPI.

3.3.3.2 Extracting items from requirements files

Some projects install Python dependencies from requirements files. Typical naming conventions are `requirements.txt` or `requirements-<env>.txt`, where env may be an environment like production or development. We look for and parse all requirements files which meet either naming convention, extracting all package specifiers within the project's requirement files using the most common subset of the requirements format specified by PEP 508. Any discovered packages are validated to exist on PyPI, and, if they exist, included in the set of pip packages.

3.3.3.3 Transaction format

Parsed dependencies from each project are converted into an intermediate transaction format for association rule mining. Each project is considered as a single transaction, and its package dependencies are written as a space separated line. Package names are prefixed with the name of the package management system, either `apt_` or `pip_`. This is both to prevent name collisions and preserve system information while generating association rules.

For example, the following Dockerfile

```
1 FROM python:2.7.13
2 COPY snippet.py /snippet.py
3 RUN ["apt-get","update"]
4 RUN ["apt-get","install","-y","libmemcached-dev"]
5 RUN ["pip","install","pylibmc"]
6 CMD ["python","/snippet.py"]
```

is parsed as the transaction

```
1 apt_libmemcached-dev pip_pylibmc
```

3.3.3.4 Rule generation

Association rules are generated from transaction data using the apriori algorithm implementation from the R package arules.[5] We use the default minimum confidence level of 0.8. Rules are restricted to those with a maximum length of two, meaning one package in the antecedent and one package in the consequent. Minimum support is chosen to restrict to itemsets where at least three examples are seen in the transaction data. The support level was chosen to filter itemsets that occurred randomly.

[5] https://cran.r-project.org/web/packages/arules/arules.pdf

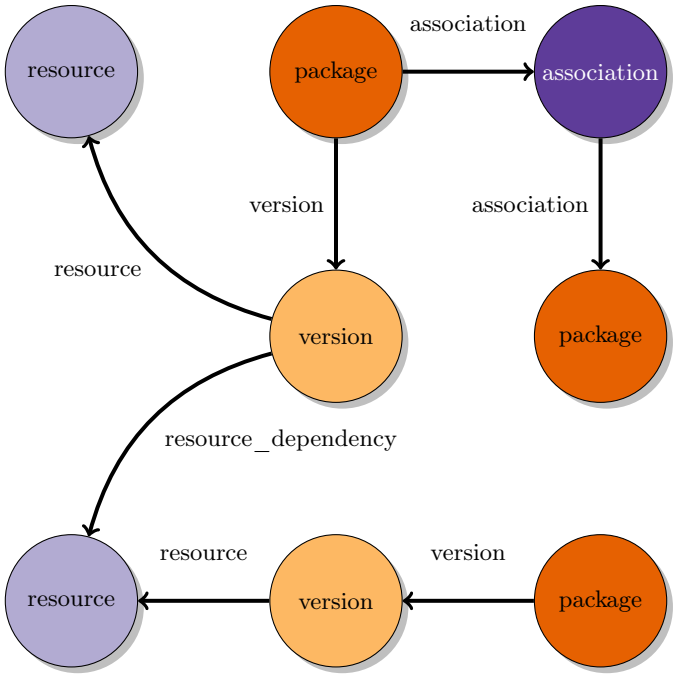


Figure 3.2 Relationships between the resource nodes in the DockerizeMe inter-dependency graph.

3.4 Knowledge Representation

We model the knowledge base as an inter-dependency graph [Ger07], depicted in Figure 3.2, stored using the Neo4J graph database. Nodes represent existing objects in the knowledge base, and directed edges represent the relationships between them. We now describe the nodes and edges used in the inter-dependency graph schema.

3.4.1 Package Nodes

Each unique package known to DockerizeMe is stored as a node in the inter-dependency graph. Package nodes are tagged with the label `package` and store both the package's name and package management system. We enforce that the name and package management system be unique together. That is, no two packages served by the same system may have the same name.

3.4.2 Version Nodes

All known versions of a package are represented as a version node. Versions are tagged with the label `version` and store the package version number. Additionally, a directed `version` edge connects the package to its version.

3.4.3 Resource Nodes

A resource is one of the directly importable package objects discovered during static analysis. Nodes are tagged with the label `resource` and store the object name. Because a package's resources may change between versions, resource nodes are owned by version nodes. This is indicated by a directed edge from the version node.

3.4.4 Modeling Dependencies

Dynamic analysis discovers a package version's dependencies on external resources. We model this relationship in the knowledge base as a directed `resource_dependency` edge from the version node to a resource node matching the name of the required resource.

3.4.5 Association Nodes

Association nodes represent individual association rules. Nodes are tagged with `association` and maintain metadata for `confidence`, `support`, `lift`, and `count`. Directed `association` edges connect packages with their association rules. A `package` $\rightarrow$ `association` edge means that the package is in the rule antecedent, and an `association` $\rightarrow$ `package` edge means the package is in the association rule consequent.

3.5 Inference Algorithm

When given a target application, the goal of inference is to determine all dependencies required to run the application without error. There is an additional constraint that dependencies must be returned in a correct order. For example, if some package A depends on B and C at install time and B also depends on C at install time, then the correct order of installation must be C, B, A.

The inference algorithm first extracts imported resources from the target application (Section 3.5.1) and interrogates the knowledge base to determine the set of packages that the resource likely belongs to (Section 3.5.2). The inter-dependency graph is then traversed starting at each of these root nodes to determine transitive dependencies (Section 3.5.3).

3.5.1 Parsing Target Code

The first task is to parse a target application and extract a list of all imported resources. We do this by building an abstract syntax tree (AST) of the source code. In Python, imported resources are defined by `Import` and `ImportFrom` nodes, which correspond to the statements `import <package>` and `from <package> import <resource>`. The full resource name at an `Import` node is the node name. For `ImportFrom` nodes, the full resource name is the name of the module plus the names of each resource being imported. Algorithm 1 outlines pseudocode for visiting both node types in the AST.

```
Procedure VisitImport(node)
if not IsStandardLibrary(node) then
    import_libraries += name(node)
end
Procedure VisitImportFrom(node)
if not IsStandardLibrary(node) then
    for resource in resources(node) do
        import_libraries += (
            name(module(node)) + name(resource)
        )
    end
end
```

Algorithm 1: Procedures for parsing imported libraries from Import and ImportFrom node types in a Python AST.

Resources are filtered to exclude those in the standard library. We perform filtering by checking to see if the resource exists in a clean Python environment with Python import tools located in the `imp` module. Pseudocode is provided in Algorithm 2. First, we look up the file system path of a resource by the resource name. If path lookup fails with an `ImportError`, we know that the module cannot possibly be part of the standard library. If lookup succeeded, we verify that the resource name either matches that of a known Python builtin or that the resource path does not contain `site-packages` and `Extras`. Extra packages are sometimes included with a Python distribution, but are not a part of the standard library, and `site-packages` is where pip places other installed packages by default.

```
Procedure IsStandardLibrary(module)
    path = getPathOrError(module)
    return isBuiltin(module) or (
        'site-packages' not in path
         and 'Extras' not in path
    )
```

Algorithm 2: A Python module is considered part of the standard library if it is a builtin or it isn't installed in a location reserved for third-party packages.

3.5.2 Mapping Resources to Packages

Once the resources for an application are known, they must be mapped back to a set of packages that can be installed. We perform this reverse lookup by querying our knowledge base and the package management system of record for potential matches. A match between a resource required by an application and an installable package may be determined by a full or partial

match on one or more known resources in the knowledge base, or a full match on a known package (either in the database or through the package management system).

```
Procedure MapResourcesToPackages(resources)
    packages = set()
    for resource in resources do
        packages += queryKnownResourcesStartingWith(
                    name(resource))
        packages += queryKnownPackagesByName(
                    name(resource))
        if not containsByName(packages, name(resource)) then
            packages += queryPackageManagementSystem(
                        name(resource))
        end
    end
    return packages
```

Algorithm 3: Mapping a list of resources imported by a code snippet to a list of packages to which they may belong.

A full match on a resource queries the knowledge base for any known resources whose names exactly match the name of a resource used by the application. If any such resources are found, the packages that own them are returned. We filter package results to be distinct, as it may be the case that some package has multiple versions with the same resource.

Partial matches search for any known resources where the name of a resource used by the application starts with the name of the of the resource in the knowledge base. For example, consider the simple Python script below:

```
1  import zope.interface
2
3  class Interface(zope.interface.Interface):
4      attr = zope.interface.Attribute('Attribute')
5
6  print(type(Interface))
```

The script imports the resource `zope.interface`, which corresponds to a package on PyPI by the same name. However, the `zope.interface` package has a top level resource of the name `zope` with a submodule named `interface`. The partial match ensures that packages whose resources follow this naming convention get matched when performing reverse lookup. Looking for exact matches is a special case of partial matches where the full resource names are equivalent.

Additionally, we check to see if any package exists with the same name as a required resource. Previous work showed that this happens approximately 45% of the time, and doing so may result in finding the correct package for a resource even if its resources could not be discovered through static analysis [Hor18].

When reverse lookup has completed, package names are normalized to match the name on the package management system, as some systems treat certain characters as being identical. For example, according to PEP 508 [Col], Python does not distinguish between a dash and an underscore, so `flask_heroku` matches the package `flask-heroku`.

3.5.3 Transitive Dependency Recovery

Knowing only the packages corresponding to top level resources is often not sufficient for correct environment configuration, as those packages may themselves depend on other packages. Information about the transitive dependency of packages is encoded in the inter-dependency graph through `resource_dependency` and `association` relationships. Assuming the inter-dependency graph contains all necessary relations, the set P of packages which must be installed is the set S of resolved direct dependencies (Section 3.5.2) unioned with the set R of packages reachable from S.

It is, however, not sufficient just to compute P. We must also preserve a correct ordering of dependencies such that each package is installed before any other package which depends on it. We do this by performing a depth first search rooted from each package $p \in S$, where neighboring packages are computed by following the directed `resource_dependency` and `association` relationships. An ordered list of packages is maintained throughout the DFS, and a package p is added to the list once all of its children have been traversed. As in our lookup procedure, the names of each package are normalized using the specific package management system they belong to.

In an acyclic graph, the ordering returned by our DFS based transitive dependency resolution results in a reverse topological order. However, we cannot guarantee that our inter-dependency graph is acyclic. It may be the case that two packages (either correctly or incorrectly) depend on each other. Additionally, if two packages p_1 and p_2 are frequently used together, our association rule mining may have generated the rules $p_1 \rightarrow p_2$ and $p_2 \rightarrow p_1$. We use our DFS resolution as a heuristic.

3.6 Evaluation

Gistable ([Hor18]) introduces the Gistable dataset, a collection of 10,259 single-file Python scripts mined from GitHub's public gist service. Gists in the dataset were discovered by querying the GitHub gist search UI for gists in the Python language with at least one star rating and automatically scraping the returned results. Analysis of the Gistable dataset showed that the most common exit status of gists in the dataset was an import error, occurring 52% of the time. In particular, the import errors in 2,891 gists could not be fixed by Gistable's naive approach of attempting to install a package named for each of the imported resources not in the standard library. We focus on these "hard" gists for which Gistable's naive inference algorithm was not sufficient, referring to them as HG2.9k.

```
Procedure RecoverTransitiveDependencies(packages)
    encounteredPackages = set()
    dependencies = list()
    Procedure DFS(node)
        if node in encounteredPackages then
            return
        end
        encounteredPackages += node
        directDependencies = (
            queryPackageDependencies(node)
        )
        for dependency in directDependencies do
            DFS(dependency)
        end
        dependencies += node
    while root = first(packages) do
        DFS(root)
    end
    return dependencies
```

Algorithm 4: Depth-first search used to discover transitive dependencies in a correct order.

3.6.1 Methodology

To evaluate DockerizeMe, we analyzed its ability to remove import errors from gists in HG2.9k. The first step in the analysis of each gist was to use DockerizeMe to perform dependency resolution to generate a list of packages P to install. We then installed each package $p \in P$ using the Python package manager pip in a clean Python 2.7.14. The execution result of each gist was recorded as the name of the exception raised when run, if any, or `Success`. We consider a gist to be fixed in the context of dependency resolution if its exit status is anything other than `ImportError`.

Installation failures encountered while evaluating a gist were ignored. The rationale for this choice being that if the failed package is a direct or transitive dependency of the gist, execution will raise `ImportError`. However, the package may not necessarily be a true dependency, due to the nature of the association rules in the DockerizeMe knowledge base. If so, it is acceptable to continue without the package.

All gists were evaluated in a Docker container based off of the `python:2.7.14` image. The DockerizeMe Docker image configured the aptitude package manager to go through a local proxy, for efficiency, and configured pip by disabling filesystem cache and setting the default timeout to 10 minutes. Analysis jobs were scheduled on a Nomad cluster, a process and container management system that natively supports scheduling Docker containers. Compute nodes were running Ubuntu 16.04 with 4 CPUs and 8GB memory.

3.6.2 Results

We rank the quality of an inferred environment primarily by whether or not its corresponding gist experiences an import error during execution. For successful inferences, we also consider the characteristics of the inferred environment by number of direct and transitive dependencies and the total number of dependencies overall. We then address installation failures encountered in inferred environment configurations and reasons why the inference procedure may have failed to produce a working environment. Finally, we consider the exit status of fixed gists as an indication of future issues to address in automated environment configuration.

3.6.2.1 Inference

We evaluated DockerizeMe's inference procedure on a total of 2,891 Python gists from HG2.9k, a subset of the Gistable dataset chosen as a baseline that cannot be fixed by installing each package by its resource name, the natural first step taken by developers when attempting to resolve import errors [Hor18]. Our evaluation found that, of the environments generated by installing the dependencies discovered by DockerizeMe's inference procedure, an additional 892 gists (31%) executed without experiencing Python's `ImportError`. This result demonstrates an important step in automated inference of environment configuration. We discuss future improvements in the next section below.

3.6.2.2 Environment Characteristics

On average, the gists from HG2.9k for which DockerizeMe generated a working environment specification import 2–3 unique resources, consistent with the overall average for all gists in HG2.9k. DockerizeMe reported performing at least one name resolution for the top 79% of the gists, meaning that most gists import a resource which was mapped to a package with a different name. 40% had at least one transitive dependency. Overall, imported resources were mapped to an average of three direct dependencies with an additional two transitive dependencies being found by the inference procedure. Transitive dependencies were most likely to be additional Python packages.

3.6.2.3 Size

The largest environment specification proposed by DockerizeMe contained 206 unique Python and APT packages. Python packages held a larger share of the total number of packages installed, with a maximum of 195 being installed in an environment versus 48 APT packages, respectively. Comparatively, the largest environment specification among those that fixed their gist's import errors only contained 87 unique packages, with the maximum of 80 Python package installed and 9 APT packages being installed in any environment.

In both cases, the large environment size was an outlier, with most working configurations being over 17× smaller. The size of the largest working configurations suggests the need for a

post-inference reduction process capable of reducing a set of proposed packages to a minimal set which still resolve a script's import errors.

3.6.2.4 Frequency of Installation Failures

Our evaluation procedure ignored installation errors for inferred dependencies under the assumption that if a package failed to install, but the overall configuration still worked, the failed package can be removed from the final configuration. Only 9 of the 892 fixed gists (1%) experienced an installation error while building the inferred environment but still exited without experiencing an import error. Manual inspection revealed that 4 of the failures were due to an unknown dependence on a Python package and 1 was due to an unknown dependence on a system package. The remainder failed either due to issues with the packages themselves or due to network noise.

3.6.2.5 Remaining Reasons for Failure

1,999 gists still exited with Python's `ImportError` after applying inference to all gists in HG2.9k. We performed an additional qualitative coding process on a random sample of 30 such gists, along with the generated environment specification, to determine why inference failed to generate a working environment.

While performing coding, each gist was inspected to determine the root cause of its exit status. We then reran DockerizeMe's inference algorithm, inspecting the inference process at runtime to determine why the root cause was not repaired. We coded each gist according to the ultimate reason why a correct environment specification was not generated. We employed negotiated agreement during the coding process to address the reliability of coding [Cam13]. Using this technique, the first and second authors work collaboratively to clarify definitions of codes and reach agreement on the assigned code.

The most common reason for failure, occurring for 15 out of the 30 gists analyzed, was that the environment specification did not find a direct dependency. That is, the gist imported some resource that was not present after installing all inferred dependencies, indicating that there was some mapping from a resource to a Python package that DockerizeMe did not know about. Four gists failed because environment inference did not find a transitive dependency, meaning that an inferred dependency failed to install because of additional configuration requirements or that execution in the inferred environment resulted in an import error for a resource not directly imported by the gist. Another three gists required had the correct package inferred for their dependencies, but needed another version due to breaking changes made to the package's API.

The remainder of the gists failed with import error, but could not have been fixed by inferring dependencies from PyPI or APT. For example, some gists required an execution environment other than the system CPython interpreter. One such gist was written as a Sublime Text plugin and required being run from within the editor. Other gists imported local configuration files that were meant to be written by the user.

Mapping resource names to their corresponding packages remains the largest issue likely due to the size of the packaging ecosystem (DockerizeMe analyzed only the top 6% of packages and 1% of versions on PyPI when generating the knowledge base). A straightforward, though expensive, solution to the mapping problem already exists: pre-process the remaining package versions. However, given the large and continually growing environment, environment inference procedures can benefit from improved heuristics for predicting what packages may belong to a resource without prior indexing.

3.6.2.6 Exit Status of Fixed Gists

Table 3.1 Exit status of the 892 gists from HG2.9k for which DockerizeMe's inference algorithm was capable of repairing import errors, filtered to those with a count greater than 20.

Exit Status	**Count**	**Percent**
Success	473	53.0%
NameError	145	16.3%
ImproperlyConfigured	91	10.2%
IOError	41	4.6%
SystemExit	23	2.6%
AttributeError	23	2.6%
RuntimeError	23	2.6%
Other	73	8.1%

Table 3.1 shows the exit status of gists from HG2.9k which had their import errors resolved by DockerizeMe. The most common exit status is `Success`, occurring for over 50% of the gists. The next most common exit status was Python's `NameError`, meaning that the gist attempted to access an object reference which did not exist. In most cases, instances of `NameError` are an issue with the gist itself and outside the focus of DockerizeMe. An exception to this is the use of wildcard imports where the resources provided by a package have changed. For future research, we can determine with static analysis if this use case is common and in need of addressing.

Of the remaining 30%, 91 exited due to Django's `ImproperlyConfigured` error and 41 due to `IOError`. `ImproperlyConfigured` is an exception raised by the Django framework on initialization if it is missing a required configuration in the settings file. `IOError` can indicate an issue communicating with a service, such as a database server.

The most common exit codes after applying inference closely resemble those found by Gistable ([Hor18] Table I), with the exception of `SyntaxError` and its child `IndentationError`, which were filtered out by the selection criteria.

Future research can evaluate how gists rely on configuration files, environment variables, and external services with the intention of generating an environment configuration that provides

these resources.

3.6.2.7 Summary

We conclude with a summary of our results.

> Our inference algorithm recovered an additional 31% of environment configurations over the baseline approach. DockerizeMe was able to successfully resolve the correct packages when there was no direct name match, and discover transitive dependencies. Python gists often require non-trivial environment configuration in order to run. Future work is needed to handle other configuration steps, such as missing environment variables and providing expected services.

3.6.3 Inference on Real-World Snippets

The following snippets illustrate how DockerizeMe's inference algorithm overcomes challenges to provide a correct environment specification.

3.6.3.1 Package name resolution

The first challenge to overcome is determining which packages provide the resources used by some code snippet. As [Hor18] discovered, determining the correct package to install is still a challenging task for developers with experience in performing software configuration.

Consider the following snippet, which imports the resource PIL and uses it to create and save a new image.

```
1  from PIL import Image
2  img = Image.new('RGB', (100, 100))
3  img.save('image.png')
```

PIL is not a part of the Python standard library, so the package containing it must be installed. There *is* a package named PIL on PyPI. However, attempting to install it reveals that the package is not maintained and has no published versions that can actually be installed.

DockerizeMe realizes that the package PIL cannot be installed because it has no available versions. Further, it knows that another Python package, Pillow, also has a resource named PIL. The inference algorithm recommends the following Dockerfile which runs the snippet without import error.

```
1  FROM python:2.7.13
2  COPY snippet.py /snippet.py
3  RUN ["pip","install","Pillow"]
4  CMD ["python","/snippet.py"]
```

3.6.3.2 Transitive dependencies

Another challenge faced by dependency resolution is determining transitive dependencies. In some cases, a snippet can rely on a package which itself relies on one or more other packages. The following snippet demonstrates this using the module dashtable to convert an html formatted table to a GitHub markdown table.

```
import dashtable
print(dashtable.html2md("""
    <table>
        <tr><th>Header 1</th><th>Header 2</th></tr>
        <tr><td>Data 1</td><td>Data 2</td></tr>
    </table>
"""))
```

There exists a package by the name of dashtable on PyPI, and it can be installed. However, running the snippet after installing dashtable results in the error `ImportError: No module named bs4`. This is because dashtable relies on the module bs4 to parse html. Fortunately, dashtable is in DockerizeMe's knowledge base, and the inference algorithm correctly infers that dashtable relies on beautifulsoup4, the package that provides the module bs4. Running the snippet with the DockerizeMe generated dockerfile correctly results in printing the converted table.

Often, a snippet may have a transitive dependency on a package which is not served over PyPI. Consider the following snippet, which makes use of the module pylibmc. The pylibmc package on PyPI fails to compile in a clean environment because it is missing the header file `memcached.h`.

```
import pylibmc
mc = pylibmc.Client(["127.0.0.1"])
```

Association rule mining provides the correct inference here. In the configuration scripts that were parsed when building the knowledge base, any script that installed pylibmc was also likely to install libmemcached-dev using the apt-get package manager. DockerizeMe proposes a Dockerfile which installs memcached before installing pylibmc, allowing the snippet to be executed without error.

3.6.4 Limitations

While our technique can discover unspecified dependencies, there are limits to the types of inference that can be performed with our current knowledge sources. We now present each limitation and the reason behind them.

3.6.4.1 Incomplete Knowledge

DockerizeMe's knowledge base of Python packages was built by analyzing the top 10 thousand packages as listed in the Libraries.io dataset, sorted by SourceRank. Association rule mining of packages found in Dockerfiles and requirements files of public repos on GitHub was responsible for populating known apt packages.

While our process for knowledge generation is designed to target the most frequently used packages, it does not have complete knowledge of the ecosystem. The apt ecosystem has over 42 thousand packages in the default repo. PyPI has almost 150 thousand packages with over 1 million unique versions available for download.

While analyzing the full ecosystem would resolve inference failure due to not knowing about a specific dependency, such analysis would be difficult to complete within a feasible time frame. In addition, brute forcing public registries cannot inform the knowledge base about packages that are available through git or are hosted on private mirrors.

3.6.4.2 Version Inference

Knowledge base generation and the subsequent inference procedure only take into account the latest version of a package. There are, however, reasons why a code snippet may depend on another version of a package, including deprecation, removal, and renaming. Additionally, snippets and dependencies requiring a different Python version are unsupported.

3.6.4.3 Package Oriented

DockerizeMe focuses on enabling execution of code snippets with import errors by inferring their software dependencies. When this fixes import errors, the exit status is considered to be `Success`. However, the gist may have a dependency on a running service. For example, a gist which imports the database drivers for MySQL will likely attempt to connect to a MySQL server. While DockerizeMe can install the database package, it currently cannot configure and run the database service.

3.6.4.4 Non-Installable Packages

Some packages available in the Python and apt ecosystems may not be capable of being installed. For example, Python Quartz had a known bug that caused a fatal error on installation due to an improper configuration with the packaging system [Vla18]. Another package, PyTorch, was not available for install through PyPI, but the maintainers hosted an empty package on PyPI which failed installation with an informative message for the user [M.17].

DockerizeMe cannot produce a correct inference under either circumstance, as it assumes that packages discovered during knowledge acquisition are installable.

3.6.4.5 Hardware and OS Requirements

Some packages require specific hardware, like a Raspberry Pi, or a specific operating system, like Windows. DockerizeMe cannot provide a hardware configuration, and does not take into account the current configuration of the system when building a Dockerfile. In addition, the DockerizeMe knowledge base was built with system dependencies from the apt package ecosystem, and has no knowledge of system dependencies for Windows, which does not have an official package manager.

3.7 Discussion and Future Work

Although DockerizeMe's inference algorithm is capable of fixing import errors encountered in nearly a third of HG2.9k, future work is needed to address import errors in gists which still fail with `ImportError` after applying our inference procedure. Other techniques are needed for improving the quality of an inferred environment, either by improving the inference algorithm or as an additional post-processing step.

3.7.1 Exploration of Other Knowledge Sources

DockerizeMe's knowledge base is generated from the results of static and dynamic analysis of public Python packages, adding associations from publicly available Dockerfiles. These are, however, not the only knowledge sources available for use. Other potential knowledge sources include additional build or configuration scripts such as Vagrantfiles and continuous integration logs from services like TravisCI. Static and dynamic analysis using system dependence graph techniques [Lia98; Sin99] may be performed to enable inference at the package version level by detecting reliance on deprecated features or breaking changes in a package's API.

Additionally, developer generated knowledge present on sites like Stack Overflow may be usable data sources. It may be sufficient to parse accepted answers for code blocks indicating a list of packages to install, using such lists as transactions for association rule mining. Another area of research may investigate using natural language processing to parse questions related to a problem encountered during inference, then automatically apply suggested solutions in highly rated answers.

3.7.2 Breaking Cycles

We currently use DFS to return a reverse topological ordering as a heuristic, as it is only guaranteed to be correct so long as the underlying graph is acyclic. Due to the nature of the packaging ecosystem and association rule generation, we cannot guarantee that directed cycles do not occur in the inter-dependency graph.

Future research will need to focus on the prevalence of such cases, and whether or not they are responsible for issues in the build process. In cases where assumptions in our DFS implementation

result in an incorrect installation order for dependencies, we can try other methods of breaking cycles. One method might be to compute the feedback edge set of each connected component to guarantee the minimum number of back-edges are disregarded.

3.7.3 Feedback Directed Inference

A majority of the inference process is performed offline. That is, knowledge acquisition occurs prior to inference, and the inference procedure itself generates a single static configuration. This does not give inference the ability to backtrack or recover from errors in cases where an incorrect dependency is resolved, except for the naive process of ignoring installation errors.

Future research will focus on feedback directed inference, a process where inference is performed iteratively in tandem with analysis. Performing inference online will allow DockerizeMe to determine if applying a configuration results in the resolution of an import error. If it does not, inference may backtrack and attempt a different configuration instead.

Applying feedback directed inference may help improve the overall quality of environment specifications, reduce the need for post-inference minimization, and allow the inference procedure to learn as it processes gists.

3.7.4 Post-processing to reduce unnecessary dependencies

The largest working configuration inferred by DockerizeMe for HG2.9k contained 87 unique packages, while most working configurations were over 17× smaller. This suggests the need for an inference method to reduce an environment specification to a minimal working configuration.

Delta debugging may be a viable method for minimizing the environment specification. Dockerfiles produced by DockerizeMe install dependencies with a dedicated `RUN` instruction. Starting with a complete specification as produced by DockerizeMe, we can delta debug by removing `RUN` instructions until a minimal set of dependencies remain.

The challenge with delta debugging Dockerfiles, in contrast to delta debugging in a standard application repair context, is that there may not be an easily executable test suite to evaluate the quality of the environment. Even in the context of DockerizeMe, where the success criteria is informed by whether or not the executable exits with a particular status, the time necessary to build a complete environment can be prohibitive. Many Python dependencies, in addition to the time required to download, also require being compiled against header files in the local filesystem. The extra compilation step adds to the overall build time.

Docker does have the ability to cache layers for every independent stage in the build. However, the cache is only valid if the parent layer is already in the cache and the instruction string exactly matches one used to generate a child layer in the cache. This restriction greatly reduces the ability to leverage the build cache during delta debugging. It may be possible to determine the optimal order for delta debugging to maximize cache usage. We may also be able to exploit sparsity in DockerizeMe's inter-dependency graph and perform delta debugging over independent

components. If independent components are small on average relative to the size of the inferred environment, the total size of the search space can be greatly reduced.

3.7.5 Additional Languages

DockerizeMe focuses on Python, but its inference procedure and mining procedures only require that source code can be parsed for imported and exported resources. We believe our approach generalizes to languages like R or NodeJS, because such languages meet our requirements. Future research can assess DockerizeMe's ability to handle configuration for such languages.

3.8 Related Work

Cito et al. investigated the current state of the Docker ecosystem by inspecting public GitHub repositories. They found that only 66% of public Dockerfiles can be built, with most quality issues being caused by dependency issues and most Dockerfile changes being made to address build dependencies [Cit17]. The most common dependency error, according to their analysis, was the failure to pin a dependency version. While their study focuses on the buildability of Docker containers, it remains an empirical analysis and makes no attempt to repair broken configurations.

Work by Hassan and Wang has focused directly on repairing configuration build scripts. HireBuild (History-Driven Repair of Build Scripts), is designed to repair failing gradle build scripts based on potential repairs discovered from the TravisTorrent dataset [Has18]. Their work is similar to ours in that it incorporates knowledge from existing, developer driven sources and uses that information to infer a correct environment specification codified in a build script. Other approaches to configuration repair target inconsistencies between file systems and configuration scripts [Su07] or capture and replay developer changes [Wei17]. Macho et al. focus on repairing Java projects using the Maven build system by updating dependency versions, deleting listed dependencies, or explicitly specifying common repositories [Mac18].

In contrast to these approaches, our work focuses on generating a complete environment specification for a codebase without a prior configuration or developer input. To our knowledge, this is the first successful use of analysis to infer environment specifications entirely from scratch.

Related to our proposed future investigation of applying delta debugging to find a minimal environment specification is Cimplifier, a technique for partitioning a container's resources into a set of smaller, independent containers [Ras17a]. A natural side effect of the Cimplifier process is container slimming due to only maintaining the resources which are used during application execution. A similar process may be a reasonable alternative to delta debugging, but may suffer from the limitations of dynamic analysis to uncover all dependencies.

3.9 Conclusion

We investigate a technique for the automatic inference and configuration of a computing environment capable of executing an arbitrary Python code snippet without resulting in an import error. Our technique builds a knowledge base of known Python packages. It discovers information about package dependencies using a combination of static analysis, dynamic analysis, and association rule mining. Dependencies are resolved in a correct order for installation. Finally, we provide a tool, DockerizeMe, which delivers an environment configuration as a Dockerfile for the Docker container system. Results from our study showed that DockerizeMe resulted in a 30% improvement over a baseline approach to environment configuration.

DockerizeMe is a fist step in automating environment configuration, a process with the potential to save developer time and effort, as well as reducing potential mistakes in application deployments. While we focus on enabling the executability of Python code snippets, we believe our inference procedure can extend to other languages with a package management ecosystem. Future research will focus on improving the inference procedure and investigate support for other configuration properties such as environment variables or external services.

CHAPTER

4

DEALING WITH VERSIONS

This work originally published as Horton, E. & Parnin, C. "V2: Fast Detection of Configuration Drift in Python". *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 477–488

4.1 Introduction

Code snippets are commonly used by developers to provide API documentation [Par13] and act as examples for learning and reuse [Wan15; Hor18; Rul18]. Stack Overflow, a question and answer site for developers, has over 1M questions relating to Python, a popular and fast growing programming language [Wan15]. On GitHub's gist system, developers have shared over 300K public Python code snippets [Hor18]. Both feature prominently in search results for API documentation [Tre18], and a study by Yang et al. found over 4M code blocks from Stack Overflow snippets that had been reused in public GitHub projects [Yan17]. Recently, code snippets in Jupyter notebooks have become a standard for sharing and replicating scientific work and more [Rul18].

Unfortunately, many code snippets require a non-trivial environment configuration in order to execute successfully [Yan16; Sul16], and are not accompanied by sufficient information for developers to easily recreate that configuration [Hor18; Hor19a; Pim19]. This leads to the problem of configuration drift, where a code snippet goes out-of-date because the APIs that it depends

on experience breaking changes over time. Developers struggle to determine if a code snippet has experienced configuration drift, or is simply incorrect [Hor18]. In some instances, a breaking change made to an API may be highlighted in release documents made available to developers. In the worst case, determining if and when a breaking change was made requires an exhaustive search through previous dependency versions.

The ability to validate and detect out-of-date code snippets is the most important consideration for quality of code reuse according to a recent survey of 183 software developers [Wu18]. Pimental et al. found that only 24% of Jupyter notebooks could be executed, and only 4% had reproducible results [Pim19]. They note that reproducibility suffers because the notebook format does not encode dependencies or dependency versions. For example, the recent release of Tensorflow version 2.0 introduced many breaking changes. As a result, many Jupyter notebooks are not runnable with the latest version of the framework. This proves detrimental, as developers have remarked that documentation for the Tensorflow platform is largely based on examples.[1] Overall, there is an unmet need for checking if code examples are up-to-date and runnable.

This work presents V2, available at `https://github.com/v2-project/v2`, a tool that determines if a code snippet is out-of-date by detecting configuration drift. V2 is based on the observation that an instance of configuration drift often manifests as a crash during execution. It can therefore be categorized by the illuminating failure (crash) paired with a configuration patch sufficient to enable execution. Patches consist of changes to dependency versions, and act both as a certificate of correctness for the snippet and enable execution with an older configuration if desired.

Unlike other work in configuration repair [Mac18; Has18], V2 automatically infers up-to-date candidate environment configurations from a code snippet without the need of developer input or pre-existing build scripts. If the code snippet experiences a crash when executed in its candidate environments, V2 searches for configuration drift by employing *feedback-directed search*, a search strategy that incorporates feedback from code snippet execution and knowledge about prior API breakage events to prune and prioritize configuration patches from the space of all possible environment configurations.

We show that V2 is able to discover configuration drift in open source Python code snippets from the Gistable dataset [Hor18] and from a wide array of Jupyter notebooks [Rul18]. For both datasets, we show that incorporating execution feedback and knowledge of previous API breakages enables V2 to more quickly decide when configuration drift is present in a snippet.

4.2 Motivation

Consider the code snippet fragment presented in Figure 4.1a. The fragment contains the import statements for `Lasagne` and `Theano` from a public Python gist found on GitHub. Both libraries can be found on and installed from the Python Package Index (PyPI). However, after installing

[1] https://news.ycombinator.com/item?id=18445225

```
import math
from functools import wraps

from theano import tensor as T
from theano.sandbox.rng_mrg import MRG_RandomStreams as RandomStreams

from lasagne import init
from lasagne.random import get_rng

...
```

(a) Import statements for the Lasagne and Theano APIs extracted from GitHub Gist a003ace716c278ab87669f2fbd37727b.

```
FROM python:3.7
ADD bayes.py /bayes.py
RUN ["pip", "install", "Theano==1.0.4"]
RUN ["pip", "install", "Lasagne=0.1"]
CMD python /bayes.py
```

(b) A base environment configuration, using the Docker container system Dockerfile format, containing the gist and the latest versions of Theano and Lasagne. Due to configuration drift, the gist does not execute successfully using the configured versions.

```
DecrementSemverMajor(Theano==1.0.4) -> 0.9.0
DecrementSemverMinor(Theano==0.9.0) -> 0.8.2
```

(c) Environment mutations made while searching for the correct version of Theano.

Figure 4.1

version `0.1` and `1.0.4` (the latest versions), respectively, executing the code fragment with Python 3 results in the following exception:

```
...
File "/usr/local/lib/python3.7/site-packages/lasagne/layers/pool.py", line 6, in <module>
  from theano.tensor.signal import downsample
ImportError: cannot import name 'downsample' from 'theano.tensor.signal'
↪  (/usr/local/lib/python3.7/site-packages/theano/tensor/signal/__init__.py)
```

The exception indicates Lasagne has attempted to import `downsample` from Theano and failed because the current version of Theano does not provide it in the expected location. The Lasagne installation documentation[2] indicates that the library is tightly coupled with Theano, and that a very recent version of Theano is often required to run correctly:

> Lasagne has a couple of prerequisites that need to be installed first, but it is not very picky about versions. The single exception is Theano: Due to its tight coupling to Theano, you will have to install a recent version of Theano (usually more recent than the latest official release!) fitting the version of Lasagne you choose to install.

However, the current version of Lasagne was released before the current version of Theano. While the documentation indicates that the correct configuration requires installing a pre-release version of Theano, it may actually be the case that Lasagne has drifted and is now out-of-date. The Theano versioning scheme uses semantic versioning[3] (semver), which tells us the current version of Theano is a major version with some patches. We start searching for the correct version by going back to the latest release of the previous major version, `Theano==0.9.0`. When that fails, we go back an additional minor version, `Theano==0.8.2`. Using this version allows the gist to be executed successfully, although with a warning that the "downsample module has been moved to the theano.tensor.signal.pool module," confirming that execution originally failed because the code had fallen out-of-date.

4.3 V2

V2, *"Version 2,"* is an extension of DockerizeMe [Hor19a] that adds support for validating multiple environments and reasoning over dependency versions. V2 determines if a code snippet or its dependencies is out-of-date by discovering instances of configuration drift. Each instance of configuration drift is identified by two things: 1) a runtime failure and 2) a configuration patch consisting of version changes.

To detect configuration drift, V2 first generates candidate environments that are potentially correct. That is, the environment is configured with the dependencies that V2 infers as being potentially necessary for executing a snippet correctly. It then executes the snippet in this

[2] https://lasagne.readthedocs.io/en/latest/user/installation.html
[3] https://semver.org/

inferred environment, recording any execution failures as they are encountered. For each new execution failure, V2 applies mutations to installed dependency versions to generate a new candidate configuration until the fault is either resolved, or no new candidate configurations can be made. V2 uses information about snippet execution and prior API breakage events to prune and prioritize candidate configurations. We now detail each stage of the repair algorithm in detail.

4.3.1 Candidate Environment Generation

In order to find instances where configuration drift has caused a failure, V2 must first have one or more candidate environment configurations for a code snippet. These should contain the set of required dependencies, although they may not have the correct versions. V2 uses the Docker container system to specify configurations for isolated environments. The use of Docker guarantees a clean and consistent base environment for V2 to work in, and allows each candidate environment to be based off of a standard image.

Although the scheduled end-of-life for Python 2 is 2020 [Pet], many older scripts will not run in Python 3, as they use syntax that was removed from the language. Candidate generation must first determine which language runtime to use: either Python 2 or Python 3. The correct language runtime is detected by attempting to parse the code snippet using Python's built in AST module for each language runtime (2 or 3). If the code snippet parses in both language runtimes, then two potential candidate environments are generated.

V2 then uses the dependency resolution algorithm and knowledge base provided by DockerizeMe [Hor19a] to populate each candidate environment with an initial set of dependencies in the following manner. V2 extracts fully qualified names of resources appearing in import nodes of the AST. Imported resources not in the Python standard library are mapped to a set of packages from the Python Package Index (PyPI) that potentially provide them, although the set may be an overestimate. These packages are considered the direct dependencies of the code snippet. V2 also resolves all transitive dependencies, i.e., packages depended on by a direct dependency or by another transitive dependency. Each dependency is initially pinned at the latest version available on PyPI, because DockerizeMe does not consider versions, and a final installation order is generated such that, for each dependency, it is installed only after all packages that it depends on.

A $(runtime, dependencies)$ tuple defines a candidate environment specification containing the correct language runtime level and dependencies listed in a valid installation order.

4.3.2 Environment Validation

The V2 validation phase accepts as input a single code snippet and a candidate environment specification in the $(runtime, dependencies)$ format. It returns a status code indicating the result of executing the code snippet in the candidate environment, plus any metadata about the

execution or encountered failures. Validation consists of two phases: 1) environment configuration and 2) snippet execution.

During the configuration step, V2 creates a new execution environment using a Docker container based on the language runtime. This guarantees a consistent starting environment. It then configures the execution environment according to the candidate environment specification by installing each dependency in order. Installation failures are recorded as part of the validation metadata, but do not halt validation. If a dependency which is not required fails to install, it will not affect the result of execution. Conversely, if a required dependency fails to install, it will produce an execution failure.

Finally, the snippet is executed in the configured environment. If the snippet runs to completion without experiencing a failure, the execution result is coded as `Success`. If an exception is encountered during execution, the snippet is considered to have failed validation and the execution result is coded as `Exception`. In this case, the exception name, message, and stack trace are provided in the validation metadata.

In certain cases, a code snippet will run indefinitely instead of exiting. For example, a snippet which waits for user input will block forever. If a snippet has not exited after a time limit, execution is stopped and the validation is coded as `Timeout`. A timeout is considered neither a success nor a failure, as it is generally undecidable if the snippet would have eventually exited. Regular Python snippets are run with a time limit of one minute. Because they generally involve computation and require a longer runtime, Jupyter notebooks are run with a base time limit of two minutes, plus an additional minute for each cell in the notebook.

4.3.3 Environment Mutation

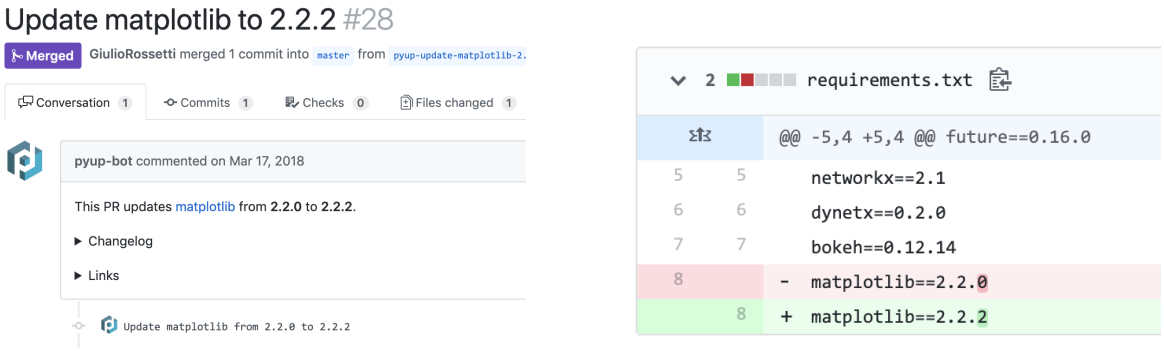
Configuration drift is classified by a validation failure and a patch. Each patch consists of a sequence of configuration changes (mutations) to the environment configuration which are sufficient to resolve the fault. V2 supports two classes of mutation operators for dependency versions. The first class is based on the semantic versioning scheme, and the second class is based on pre-existing knowledge about version changes.

4.3.3.1 Semantic Versioning

Semantic versioning (semver) is a versioning scheme which defines major, minor, patch, and prerelease changes to a package. Breaking API changes must be accompanied by a major version change, while backwards compatible additions are accompanied by a minor version change. However, before version `1.0.0`, semver defines that an API should not be considered stable, and that anything may change at any time. Many Python dependencies follow a versioning scheme that is, or can be interpreted as, semver.[4]

V2 defines two mutation operators for semantic versions: 1) `DecrementSemverMajor` and 2)

[4]https://www.python.org/dev/peps/pep-0440/



(a) Pull request to upgrade matplotlib from 2.2.0 to 2.2.2.

(b) Git diff.

Figure 4.2 An upgrade event initiated by the upgrade bot PyUp.

`DecrementSemverMinor`, which mutate the major and minor version level of a dependency in an environment configuration as seen in Figure 4.1c. Both mutation operators will decrement the specified version level by one if possible, but will choose the latest release at that version level. For example, if a package has versions `2.1.0`, `1.2.0`, and `1.1.0`, then `DecrementSemverMajor(2.1.0) -> 1.2.0`. `DecrementSemverMinor` only operates within a major version, it will not roll back to a previous minor version if doing so also requires decrementing the major version.

4.3.3.2 Upgrade Matrix

In the worst case, all versions of a dependency may be enumerated by combining the semver mutation operators described in Section 4.3.3.1. However, such a brute force approach is generally not feasible due to the large number of potential candidate environments in the version configuration space. A more efficient approach is to prioritize or prune environment configurations based on how likely they are to fix a validation failure.

To make this estimation, we extrapolate from TravisCI build statuses of Python projects that experience version upgrade events. An upgrade event occurs when a commit is made that upgrades the version of a single project dependency, triggering a TravisCI build for the project. Figure 4.2 shows an upgrade event initiated by `pyup-bot`, an upgrade bot for Python projects.

All upgrade events were discovered by mining Google BigQuery for GitHub pull requests from January 2014 though January 2019. We limit our search to pull requests which change the version for only a single Python dependency and were submitted by an account with the name `dependabot[bot]`, `dependabot`, `snyk-bot`, or `pyup-bot`. We then extract the project build statuses for the original and new dependency versions from Travis CI, excluding build histories where one of the statuses was either canceled or errored. In total, this provided 7,104 upgrade events for 193 distinct packages that experienced at least one failure.

Taken together, the build statuses for a single package form a version upgrade matrix for that package, where each row and column indicates upgrading from one version to another. Each

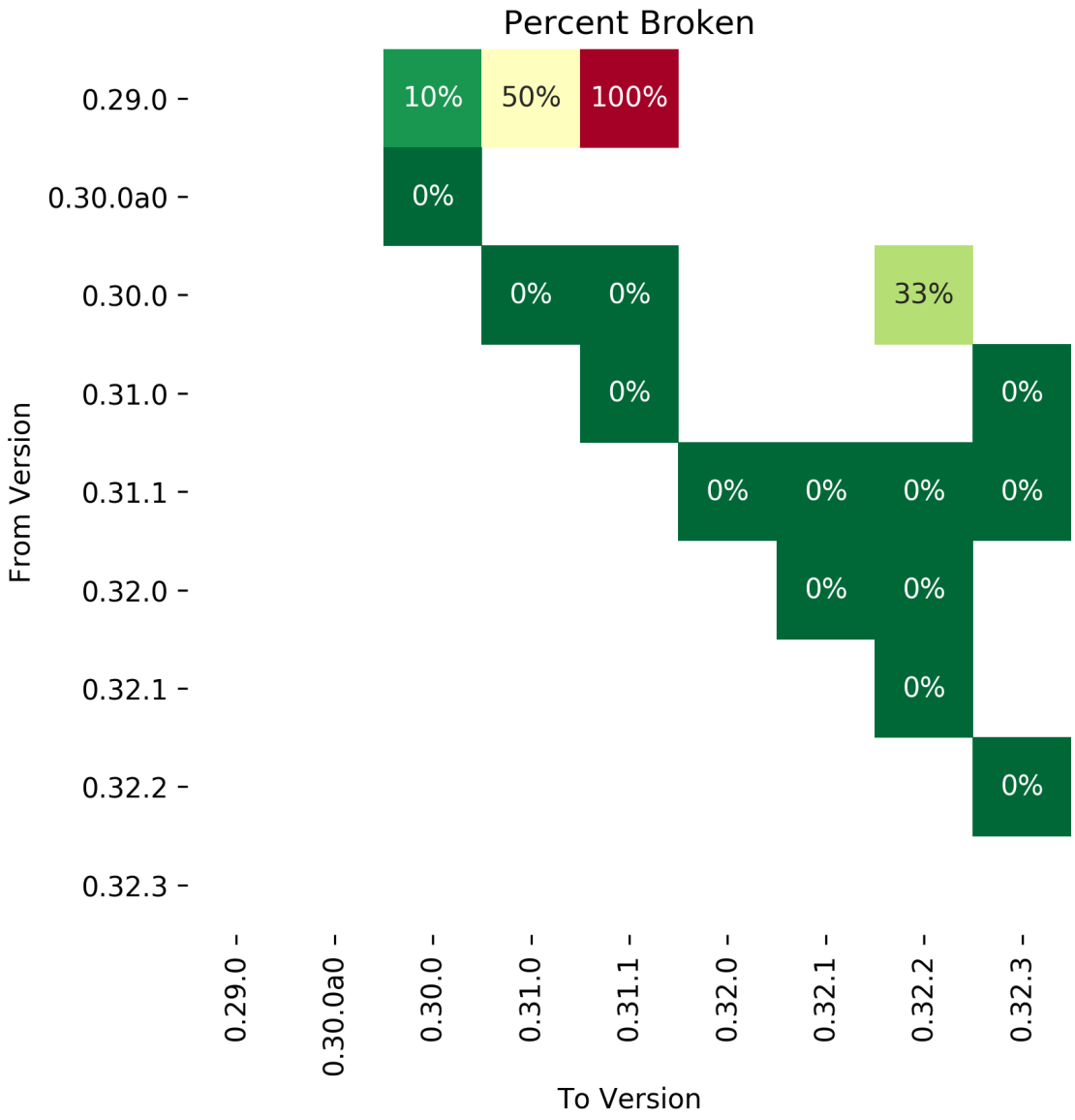


Figure 4.3 Version upgrade matrix for Wheel.

cell of an upgrade matrix contains the percent of builds that were broken by performing the version upgrade, where a breakage is determined by a build status that changes from passing to failing as a result of the upgrade. Figure 4.3 shows the version upgrade matrix for the Python package Wheel. An incompatibility cluster in the top row indicates that upgrading from version `0.29.0` becomes more likely to cause a failure as the distance between versions increases.

Intuitively, we apply the heuristic that upgrades which break a larger number of builds are more likely to indicate a backwards incompatible change. Conversely, given a failed validation, downgrading to a version which experienced a large number of upgrade failures is more likely to fix the fault. When V2 finds a validation failure caused by a dependency with an upgrade

matrix, it can leverage information about the incompatible upgrades to test only those versions likely to result in a fix, pruning the rest of the version space.

4.3.4 Feedback-Directed Search

The problem of finding an instance of configuration drift can be solved by conducting a search through the full configuration space for a code snippet. We model the configuration space as a configuration graph $G = (V, E)$, where each vertex $v \in V$ is a potential environment configuration and each directed edge $(u, v, m) \in E$ is a pair of configurations (u, v) and a mutation m such that applying mutation m to configuration u results in configuration v. We now formally define the problem, DriftSearch, of finding patched configurations in G that fix faults caused by configuration drift. If some configuration fixes all faults, we say that it is a working environment configuration.

DriftSearch

Input:	A configuration graph $G = (V, E)$ and a starting environment v.
Problem:	Find a working environment w.
Objectives:	Minimize the distance from v to w. Minimize the number of vertices explored.

We implement a search strategy, *feedback-directed search*, that generates a configuration space from a candidate environment configuration by applying the mutation operators defined in Section 4.3.3. Feedback-directed search incorporates information about the executability of a code snippet within candidate environments to intelligently drive exploration of the configuration space, reducing the number of environments in the configuration space explored and allowing V2 to quickly converge to a working configuration.

V2 begins feedback-directed search with a list of candidate environment configurations generated as described in Section 4.3.1. If more than one candidate environment was generated, feedback-directed search operates on the environment specifications round-robin, exploring the configuration space for each candidate in tandem. Doing so allows feedback-directed search to satisfy the objective of minimizing the distance, or number of mutations, from starting environment v to working environment w in the case that one candidate environment is easily fixable, but the other is not. The search algorithm is structured in three distinct phases: validation checkpointing and pruning, fault localization, and exploration. A high-level implementation of feedback-directed search for a single environment is outlined in Algorithm 5. We now highlight each phase in detail.

4.3.4.1 Validation Checkpointing and Pruning

Given a code snippet s and candidate environment c, feedback-directed search first performs validation of s in c using the procedure outlined in Section 4.3.2. If the candidate environment

```
Procedure FeedbackDirectedSearch(environment)
    checkpoint = null
    do
        // Validation
        validation = validate(environment)
        if fixed(checkpoint, validation) then
            record_drift(checkpoint)
            checkpoint = validation
            if not fixable(checkpoint) then
                return environment
            end
        end

        // Fault Localization
        if updated(checkpoint) then
            dependency = localize(checkpoint)
            if dependency is not null then
                if has_matrix(dependency) then
                    mutator = matrix(dependency)
                end
                else
                    mutator = IDDFS(dependency)
                end
            end
            else
                mutator = IDDFS(environment)
            end
        end

        // Exploration
        environment = next(environment, mutator)
    while not successful(checkpoint)
    return environment
```

Algorithm 5: Implementation of feedback-directed search for a single candidate environment. Exploration is performed by either using a version upgrade matrix or iterative-deepening depth-first search (IDDFS).

produces an execution failure, the validation result is saved as a checkpoint, and c will be mutated to fix the fault. After each environment mutation, s is again validated in c.

Conceptually, at each stage of the search process, a validation checkpoint represents the latest unfixed validation failure. We consider the fault indicated by the validation checkpoint to have been fixed by a sequence of mutations to c if a newer validation results in an execution which covers more of s. That is, the mutations made to c allow execution of s to proceed past the line which previously caused a failure. Whenever such a sequence is discovered by feedback-directed search, it, and the validation checkpoint, are recorded as an instance of configuration drift, and the checkpoint is updated to the newer validation. If a validation indicates that a sequence of mutations has not changed the validation result, or that execution exits on or before the line reached by the checkpoint, it is discarded and the search process continues.

Whenever a new validation checkpoint is discovered, V2 determines if it is potentially fixable via mutations to dependency versions. If it is not potentially fixable, search halts and returns the current environment and instances of configuration drift reported. A validation checkpoint is considered potentially fixable if the execution exception does not indicate a failure due to the local file system and satisfies one of:

1. It is caused by an installed dependency.
2. It is an import error related to an installed dependency.
3. It is one of `TypeError` or `AttributeError`, which can indicate breaking changes in a public API.

4.3.4.2 Fault Localization

During search, V2 prunes the local configuration search space by performing fault localization to map validation checkpoint failures to a single dependency. If the execution failure was caused by code not belonging to the code snippet or the Python standard library, V2 extracts the last dependency from the stack trace that was installed as part of the environment configuration. If the exception is an import error related to a single installed dependency, V2 indicates that dependency. In cases where fault localization fails to highlight a single installed dependency related to the failure, V2 continues exploring the local search space.

4.3.4.3 Exploration

Whenever a validation result indicates that a code snippet s does not execute successfully in a candidate environment c, feedback-directed search searches the local configuration space for a fix by applying mutations to c. There are three exploration strategies, based on the success of fault localization and whether V2 can leverage one or more version upgrade matrices.

If V2 succeeds in localizing a fault to a single installed dependency d, and that dependency has a version upgrade matrix, feedback-directed search proceeds by querying all pairs of versions

$(v_{i,1}, v_{i,2})$ such that the version matrix indicates a nonzero percent of builds were broken by upgrading from $v_{i,2}$ to $v_{i,1}$, and $v_{i,1}$ is at most the current version of d. Selection in this manner implicitly disregards the versions which experienced no breaking upgrades. An ordering of versions is generated by sorting all pairs in descending order by $v_{i,1}$. Then, for each $v_{i,1}$, all $v_{j,2}$ not already in the ordering are sorted in descending order by percentage of builds broken and appended to the order. While validation indicates that the checkpoint has not been fixed, d is mutated to be the next version in the ordering and validated again.

When V2 is capable of localizing a fault to a single installed dependency d, but that dependency does not have a version upgrade matrix, feedback-directed search explores the local configuration space by performing iterative-deepening depth-first search (IDDFS) over the versions of d using the semver mutation operators from Section 4.3.3.1. In all cases, the operators are strongly ordered such that all instances of `DecrementSemverMajor` are applied prior to instances of `DecrementSemverMinor`. This, coupled with the fact that `DecrementSemverMinor` does not cross major version boundaries, guarantees that the same configuration is not reached at two different depths. If V2 is unable to localize a fault to a single installed dependency, feedback-directed search performs iterative-deepening depth-first search over the versions of all dependencies, making use of upgrade matrices for individual dependencies where applicable. In either case, while performing iterative-deepening depth-first search, candidate environments are only validated when they are generated.

Because of the size of the search space, tracking all candidate environments on the frontier quickly becomes intractable, limiting search algorithms which require doing so. Depth-first exploration allows feedback-directed search to generate and validate new candidate environments with smaller memory requirements. Note that, because mutation operators result in decreasing dependency versions, exploration will never undo mutations made prior to the current checkpoint. Search is halted if no sequence of mutations will lead to an as-of-yet unexplored environment.

4.4 Evaluation

We evaluate the ability of V2 to find instances of configuration drift by analyzing open source Python code snippets.

4.4.1 Datasets

Horton and Parnin previously presented Gistable, a dataset of 10,259 Python code snippets from GitHub's snippet sharing service [Hor18]. They identified a subset of approximately 2,891 "hard" gists that experienced a failure even after a straightforward approach to inferring an environment configuration. We exclude 795 gists that are known to have over 10 direct dependencies, because the large configuration space is intractable for a baseline. For example, an environment with 10 dependencies, each with 3 versions, would have 3^{10} = nearly 60K possible unique configurations. This leaves 2,096 gists that we refer to as the Gist dataset. Prior work was unable to make

Table 4.1 Summary of code snippets from datasets.

	Gist	Jupyter
Total	2,096	6,529
With a Candidate Environment	2,096	5,423
With a Successful Candidate Environment	119	758
No Successful Candidate Environment	1,977	4,665

1,999 of these gists execute successfully after candidate environment generation using the latest versions of all dependencies [Hor19a].

Second, we collect larger Python code snippets in the form of Python notebooks. Rule et al. previously collected 1.25M open source Jupyter notebooks from GitHub, making the dataset publicly available [Rul18]. As part of their release, they provide a random sample of 6,529 notebooks, many of which are written in Python. We exclude 1,106 notebooks from the sample for which no candidate environment could be generated, leaving 5,423 notebooks we refer to as the Jupyter dataset. There were several main causes for why candidate generation failed, all related to being unable to parse the snippet. Some notebook files were completely empty, or identified as using Jupyter kernels for other languages, such as R, meaning V2 could not parse the notebook source code as Python. Other notebooks contained IPython magics,[5] which are defined as statements which are syntactically invalid Python, causing difficulty in parsing.

For simplicity, and to disambiguate between the origin of code snippets, we refer to code snippets from the Gist dataset as "gists" and code snippets from the Jupyter dataset as "notebooks." Table 4.1 shows a summary of both. Together, these snippets represent a wide range of behaviors and API uses, and their generated candidate environments have over 2K unique installable packages.

4.4.2 Methodology

We use iterative-deepening depth-first search to establish a baseline for identifying configuration drift in each dataset. This baseline enumerates all states in the configuration space and represents a worst-case scenario where it is assumed a repair can be made by modifying a dependency version, but no information is available to guide configuration. For each code snippet, we record whether search finished or timed out. If search finished, we record whether or not it resulted in a successful environment configuration, and the total number of environments validated.

After running the baseline, we analyze both datasets using V2 with feedback-directed search, recording the same metrics as used in the baseline evaluation. In all cases where search finished within the timeout, we also record every instance of configuration drift identified by V2, along with the number of environments in the configuration space that were validated while identifying the instance, as evaluations are the dominating factor in cost for validation based approaches [Wei13].

[5] `https://ipython.readthedocs.io/en/stable/interactive/magics.html`

How far the final configuration is from the starting configuration is also recorded. If search did not find a fully working environment configuration, we record metadata about why the search process terminated, and which portion of the configuration space it was operating in.

We performed evaluation on a cluster of 8 virtual compute nodes running Ubuntu 16.04.5 LTS. Each compute node was configured with 8 CPU cores and 12 GB memory. Evaluations were run using the HashiCorp Nomad job scheduler, with at most 6 jobs running on a single compute node at one time. All evaluations were run with a timeout of 1 hour, after which they were given the chance to gracefully exit and record results. Evaluations which lasted longer than 2 hours were terminated. This could happen for snippets where validating the last environment took longer than an hour, as we allowed each validation to fully complete before timing out.

4.5 Results

We evaluate V2 with three metrics related to its ability to detect configuration drift: improvement over the baseline, discovery of configuration drift, and search performance.

4.5.1 Improvement Over the Baseline

The iterative-depening depth-first search baseline approach searched for a working environment configuration using only the mutation operators `DecrementSemverMajor` and `DecrementSemverMinor`. We found that the most common baseline result was a timeout at the one hour mark without finding a working environment configuration. Search failed to complete for 978 out of 2,096 gists, nearly 50%. 3,921 out of 5,423 notebooks likewise failed to complete within an hour. Comparatively, feedback-directed search only timed out on 248 gists, a reduction of 730 from the baseline analysis. This is mirrored by Jupyter notebooks, only 488 of which timed out versus the baseline of 3,903, a reduction of 3,415.

> A simple search strategy is insufficient for fully exploring the configuration space of most code snippets. Feedback-directed search aids exploration by allowing V2 to focus only on a relevant subset of the configuration space.

4.5.2 Discovered Configuration Drift

V2 discovered 175 instances of configuration drift present in 143 gists and 73 instances present in 63 notebooks. Search terminated without finding a working environment configuration for 1,882 gists and 5,017 notebooks, only 29 and 24 of which timed out, respectively (although 1,839 notebooks experienced a Jupyter error related to network issues). For 928 gists and 1,710 notebooks, search was terminated because the validation failure was heuristically determined to not be fixable by mutating installed dependency versions according the criteria in Section 4.3.4.1. In the other cases, search was terminated because V2 had explored all environment configurations

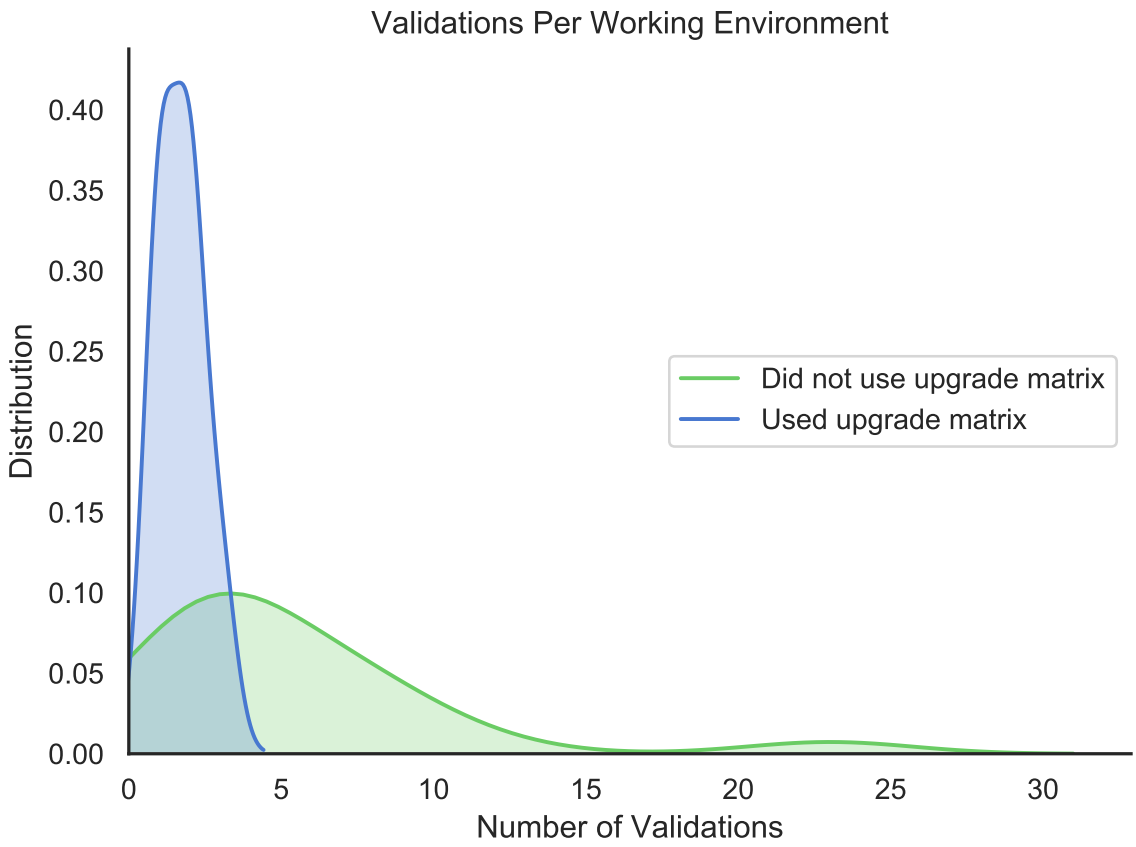


Figure 4.4 Number of validations performed on gists for which a working environment was ultimately found. If V2 is able to make use of data in a version upgrade matrix, it can converge to a working environment with fewer validations.

in the portion of the configuration space determined to be relevant to the validation failure without finding a patch.

> V2 is capable of finding instances of configuration drift in both gists and notebooks. It can potentially find more than one instance in a single snippet.

4.5.3 Search Performance

We record two metrics important to search performance, corresponding to the objectives of DriftSearch (Section 4.3.4). The first is the total number of environment configurations validated before finding a patch. This indicates the overall performance of feedback-directed search, since validation dominates runtime performance. The second metric is the number of mutations made to a candidate environment to find a patch. Both metrics reflect on V2's ability to prune and prioritize within the search space.

Feedback-directed search shows an improvement over the baseline in the average number of

validations needed to find a working environment configuration. Gists see a large improvement, dropping from 18 validations on average in the baseline to 8 when using feedback-directed search. Jupyter notebooks see a more modest improvement from 7 to 6. For gists, V2 was able to take advantage of a version upgrade matrix for 7 of the 27 (26%) that were made fully executable, bringing the average number of validations needed down from 5 to 2. Even when a working environment was not found, V2 was able to use the version upgrade matrices for about 28% of code snippets, decreasing the number of validations needed before deciding no working environment existed.

Figure 4.4 shows the distribution of number of validations required by feedback-directed search to find a working environment configuration for gists. When able to prioritize using a version upgrade matrix, V2 requires a third as many validations on average, with substantially better savings in the worst case. Time savings are also seen for individual instances of configuration drift, regardless of whether a fully working environment was eventually discovered.

A relatively small number of mutations (about 2) is required to find a configuration patch for each instance of drift on average. However, incorporating information from a version upgrade matrix results in an improvement in the worst case. For gists, the maximum number of mutations made to create a configuration patch without a version upgrade matrix was 14, while the maximum with one was 7. For notebooks, the maximum number of mutations dropped from 11 to 4. This indicates that V2 is able to correctly skip over certain versions while searching.

> Feedback-directed search is able to find configuration drift by only searching a portion of the entire configuration space. Whenever a version upgrade matrix is available, V2 can leverage the build information to prioritize dependency versions and further improve search performance.

4.5.4 Configuration Drift Found in Code Snippets

We now detail V2's behavior on two code snippets from our corpus, detailing how feedback-directed search actually found and fixed instances of configuration drift.

4.5.4.1 Prioritization of Dependency Versions

Consider the gist `moby.py`,[6] which contains the following imports from the Python library `Sphinx`.

```
1  from sphinx import addnodes
2  from sphinx.builders.html import StandaloneHTMLBuilder
3  from sphinx.util.osutil import EEXIST, make_filename
4  from sphinx.util.smartypants import sphinx_smarty_pants as ssp
```

[6] `https://gist.github.com/5866756`

Executing `moby.py` with the latest version of Sphinx (`2.0.1`), results in an import error for the module `sphinx_smarty_pants`. A developer might assume that the configuration is broken and look for a missing dependency that provides an extension to Sphinx. However, Sphinx extensions are provided from the `sphinx.ext` module, not the `sphinx.util` module. Another assumption is that a breaking change was made to the `smartypants` module.

To validate this assumption, the developer must find a previous version of Sphinx for which `moby.py` executes successfully. They start by reverting from `Sphinx==2.0.1` to the previous major version `Sphinx==1.8.5`. However, executing with this version results in the same error. They change the major version again to `Sphinx==0.6.7`, a change which also requires using Python 2, but doing so results in an import error for `osutil` on the line before, indicating that the latest version change has actually broken a part of the configuration which previously worked.

If a correct working version exists, it must be one of the other untested minor or patch level versions. Without any other information, discovering this version requires performing an exhaustive search though the `1.x` and `0.x` versions. Using a depth-first strategy, such as iterative-deepening depth-first search, the developer might additionally validate versions `1.7.9`, `0.5.2`, `1.6.7`, and `0.4.3` before finding a working configuration with version `1.5.6`.

V2, by comparison, starts by validating Sphinx version `2.0.1` and encounters the original error, localizing it to the Sphinx module. Using the available upgrade matrix, it sees that the most recent version with upgrades that resulted in a broken build was `1.7.5`, and the only broken upgrade was from version `1.4.5`. It then mutates the environment, resulting in a configuration in which the gist successfully validates.

4.5.4.2 Uncovering Multiple Instances of Configuration Drift

V2 is capable of uncovering more than one instance of configuration drift from the code snippets that it analyzes. For example, by using feedback-directed search, V2 demonstrates two instances of configuration drift in the gist `guess_candidate_model.py`,[7] which the following excerpt is from.

```
1  from sklearn.cross_validation import train_test_split
2  from keras.preprocessing import sequence, text
3  from keras.models import Sequential
4  from keras.layers import (Dense, Dropout, Activation, Embedding, LSTM, Convolution1D,
   ↪  MaxPooling1D)
5  ...
6  X = [x[1] for x in labeled_sample]
7  y = [x[0] for x in labeled_sample]
```

This gist relies on two installable packages: `scikit-learn` and `Keras`. V2 correctly generates candidate environment configuration for both Python 2 and Python 3 containing the latest versions of both of these packages.

[7] https://gist.github.com/eba2bf1a0ecd3e541146e98f35d49739

The first validation inside of a candidate environment results in a failure, returning the message "`No module named 'sklearn.cross_validation'`" even though the `sci-kit` package is included in the environment configuration. V2 recognizes this as a potential instance of configuration drift and applies the mutation operator `DecrementSemverMinor(scikit-learn==0.20.3) -> 0.19.2`.

The next validation also results in a failure, but this time with the message "`No module named 'tensorflow.'`" V2 determines that the previous failure has been fixed, because execution progressed past the line which caused the previous failure, and recognizes this as a potential instance of configuration drift involving `Keras`. In searching for a patch, it applies the mutation operators `DecrementSemverMajor(Keras==2.2.4) -> 1.2.2` and `DecrementSemverMajor(Keras==1.2.2) -> 0.3.3`.

Finally, V2 encounters the error "`name 'labeled_sample' is not defined`." It again recognizes that the previous failure has been fixed. Further, it recognizes that this failure indicates an error with the gist that cannot be fixed by modifying the environment configuration and terminates search.

4.6 Limitations

V2 presents a promising approach to finding and repairing instances of configuration drift within a code snippet and its dependencies. We show that it is capable of finding and repairing drift in a large corpus of open source Python code snippets. There are, however, some limiting factors which make it challenging to uncover all instances of drift.

4.6.1 Silent Failures

Feedback-directed search requires that configuration drift result in a crash during snippet execution, because this failure is used to guide the search process and determine when the configuration drift has been patched. This requirement is sound for a large number of breaking API changes, such as removal of modules or a change in the code signature, because such changes result in runtime or compile time exceptions. However, some breaking changes in dependency behavior may cause a failure which does not manifest as a crash. V2 cannot address such cases, although we note that the problem of determining if program behavior is expected is an open and challenging area of research [Bar15].

4.6.2 Mutation Operators and Ordering

V2 concentrates on major and minor dependency versions because they are the most likely to introduce or change behavior of an API, and its search strategy explores versions in the configuration space with equal precedence, only giving preference to configurations that are more similar to the original candidate configuration. However, developers differ in how they reason about the stability of dependencies, and communities differ on how they handle versioning [Bog15].

Other search strategies may ultimately prove to be more effective at uncovering configuration drift.

4.6.3 Overzealous Pruning

Parts of the V2 search algorithm are informed by heuristics. In cases where a heuristic erroneously produces a false positive or false negative determination, search may be terminated prematurely. For example, if a validation failure is heuristically determined to not be fixable by modifying dependency versions, but a fix actually does exist, V2 will stop exploration and be unable to report the configuration drift.

For some dependencies, search is informed by a version upgrade matrix obtained from TravisCI build results. However, an upgrade matrix may be incomplete or not sufficiently represent some breaking upgrades, causing incorrect pruning.

4.6.4 Snippet Properties

Some code snippets may be unsuitable for uncovering configuration drift. Gists can be overly general, relying on other services and requiring resources like authentication tokens [Hor18]. Failures related to these resources may shadow configuration drift that could otherwise be discovered. Notebooks are heavily used within the data science community [Rul18], and data science dependencies may be more stable than Python dependencies in general. Additionally, gists were restricted to having at most 10 direct dependencies. While this does not necessarily restrict the total number of dependencies, it may impact the distribution of results.

4.6.5 Discovery

Because exploration of the configuration space is guided by feedback from validation, V2 discovers at most one instance of configuration drift at a time. Further, validation can be an expensive process, and other instances of drift may be hidden by the most recent validation failure.

4.7 Discussion and Future Work

Recent work has shown that managing dependencies is a large and time consuming problem developers face when engaging in configuration management [Seo14; Sul16; Url18; Hor18; Hor19a]. In particular, some developers have highlighted dependency management as one of the largest problems facing Python, requiring developers to spend non-trivial amounts of time on configuration for basic tasks.[8] This problem impacts data scientists, who must overcome the friction imposed by configuration just to perform their job.[9]

[8] `https://twitter.com/jeanqasaur/status/1104990612057518081`
[9] `https://twitter.com/LibSkrat/status/1122857944675229698`

In addition, upgrading a version of a library that contains incompatibilities in data structures, signature changes to API calls, or behavior breaking changes [Rui15] can result in additional effort to address these changes. If a developer fails to understand the nature of a dependency change or performs insufficient testing, they can introduce undetected faults in code. Such factors may cause a developer to become reluctant to update dependencies. But, if they delay updating code too long, they may be locked out of being able to use important new features only available in new versions, as it becomes more and more difficult to adapt their code. For this reason, detecting instances of configuration drift has direct implications for configuration management. Being able to highlight where code contains outdated API usages, even between dependencies of that code, can help developers locate and fix usage of APIs that have been changed, benefiting work in API migration. Further, by pairing each instance of drift with a corresponding fix, we motivate code repair [Wei06], and reuse. For example, repairing Jupyter notebooks can enable replicating important calculations even in older notebooks.

Although we focus on Python, we believe detection of configuration drift is an important problem for other languages. For example, updating the MongoDB driver for Node.js from `2.0.x` to `2.1.x` affected the way order by parameters are used in the *sort* function. Previously, parameters were allowed to be specified as list of objects: `[{publish_date: -1}]` but in the newer version, list of lists must be provided: `[["publish_date", -1]]`. V2 is capable of detecting such a breaking change in any language so long as the code snippet may be parsed and validated, and previous versions of dependencies are made available.

While the technique outlined in this paper shows promise, we believe there are several areas for improvement.

4.7.1 Fuzzing

Data in version upgrade matrices is necessarily limited, as it originates from upgrades to existing open source projects which both use upgrade bots and build through TravisCI. When data does exist for a package, the matrix is usually sparse. We believe that the amount and quality of available upgrade data can be augmented. One method would be to engage in fuzzing of dependency versions for libraries. For example, the upgrade matrix for `numpy` could be filled in by choosing a set of snippets or generated tests that cover the API and observing their execution with different versions of the library. This additional execution data could help determine clusters of incompatible versions that indicate a breaking change for particular API uses.

Traditional fuzzing techniques can also be implemented to improve drift detection. Some code snippets, particularly those meant to be used as examples, define functions but do not execute them. Executing these functions with generated input would ensure greater coverage of the code snippet and its API uses, allowing further detection of instances where configuration has drifted.

4.7.2 Progression in the Face of Adversity

V2 currently only recognizes configuration drift caused by versions of installed dependencies. However, code snippets often depend on resources external to the configuration environment, such as communication with databases or REST APIs. When validation of a code snippet fails due to an external service, future work can focus on determining if the failure represents an instance of configuration drift and attempt to generate a fix. If a fix cannot be found, a "mock" patch sufficient to remove the validation failure could be inserted, allowing analysis of the snippet to continue. Missing resources, such as files on the local file system, could also be synthesized to allow code snippet analysis to continue.

4.7.3 Improvements to Search

Feedback-directed search uses heuristics and checkpointing to reduce the search space. This results in improved performance, but at the potential cost of completeness. This could be mitigated by still searching all versions, but using version matrices for prioritization. Additionally, the checkpointing process can be modified to allow for backtracking to a previous checkpoint if no solution is found (as might be the case in library version conflicts). However, V2 does not currently do so.

Finally, work can be done to improve the quality of search through the environment space. V2 assumes that the original candidate environment is complete and contains all required dependencies, if not the correct versions. We leave it to future work how best to continue searching if it is discovered that this is not the case.

4.7.4 Optimization and Improvement

The validation strategy employed by V2 currently starts by creating and configuring a new environment. This is sufficient to guarantee correctness of validation, but requires an expensive configuration phase after every mutation performed by feedback-directed search. We note that, because feedback-directed search generates a new environment configuration by applying a single mutation to the previous configuration, V2 could optimize the validation process by creating and continually mutating a single environment specification. This would greatly reduce the cost of validation.

4.8 Related Work

Previous work has concentrated on evaluating the executability of code in the context of its configuration. Sulír and Porubän find 38% of Java builds fail due to dependency related errors [Sul16]. McIntosh et al. agree that build maintenance imposes a large overhead on developers [McI11]. Yang et al. [Yan16] evaluate Python code snippets, as do Horton and Parnin [Hor18]. They find that Python code snippets are not executable by default due to

configuration errors, and Horton and Parnin additionally highlight the difficulty of correct configuration, of which inferring correct dependency versions ranks highly.

Additional studies have focused specifically on how developers manage dependencies and versions within environment configuration. Xavier, Hora, and Valente note that developers deliberately make breaking changes to APIs for several reasons related to code maintenance [Xav17]. Further, developers struggle with different options for specifying dependency versions that are supported by many package management systems [Die19; Cit17], and analysis of such systems motivates the need for better tooling to deal with problems related to versions [Dec17].

Although these works demonstrate that configuration of dependencies is a frequent cause of build and execution failure, and that developers have need of better tooling support of versions [Mir17; Xin07; Hen05; Dag11], no study attempts to uncover instances of configuration drift automatically. Horton and Parnin do classify 15 code snippets which have drifted and only execute successfully with an older version of a dependency (Table II from Gistable [Hor18]), but this classification was performed by manually generating configuration scripts.

Existing work has been performed on the automatic repair of environment configurations. Weiss et al. presented Tortoise, a system which synthesizes patches for configuration scripts, but can only do so by analyzing recorded developer actions [Wei17]. Horton and Parnin work on inferring working environment configurations without the need of developer interaction, but they crucially ignore dependency versions [Hor19a].

Work by Macho et al. [Mac18] and by Hassan and Wang [Has18] has presented automated repair of configuration scripts that includes version modifications. These are the closest related works of which we are aware. Crucially, both techniques require an existing environment configuration, and neither approach uses feedback to guide search. In addition, they focus on the Java build ecosystem, and cannot repair Python code snippets. Macho et al. prioritize versions using a distance metric that prefers small version changes. Hassan and Wang generate a ranked list of all possible patches and exhaustively apply them until some patch succeeds or there are no more patches to apply. Neither approach attempts to prune the configuration space, and prioritization is limited. There are approaches that use feedback [Nai18] and historical knowledge [Le16], but they focus on code, not computing environments.

This work presents an improvement over previous approaches by incorporating feedback and prior knowledge for more efficient identification of repairs. We also focus on identifying instances of configuration drift in code, determining when dependencies have been updated, and synthesizing a patch as a proof of existence. This is in contrast to other approaches, which focus on an existing environment configuration and assume failures result from the code being updated.

4.9 Conclusion

Motivated by the problem of detecting out-of-date code snippets, we implement V2. V2 discovers instances of configuration drift in Python code snippets, where each instance is identified by

looking for crashes that can be fixed with modifications to versions of installed dependencies.

Two techniques lie at the heart of V2's ability to uncover configuration drift. The first is feedback-directed search, a search strategy that uses information gleaned from executing a code snippet to determine which portion of the configuration space to explore next. The second is a corpus of version upgrade matrices that describe the before and after build statuses for over seven thousand package upgrade events in open source Python projects. These version upgrade matrices allow V2 to significantly prune the configuration space, and effectively prioritize the remaining states, leading to faster discovery of working configurations.

V2 found 248 instances of configuration drift in the Gist and Jupyter datasets. It was able to do so quickly, requiring many fewer validations than a baseline prioritization strategy.

CHAPTER

5

MIGRATING CONFIGURATION TASKS

Sections of this work originally published as Horton, E. & Parnin, C. "Dozer: Migrating Shell Commands to Ansible Modules via Execution Profiling and Synthesis". *44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22)*. 2022

Using Bash scripts to manage infrastructure is, according to Netflix engineer Lorin Hochstein, "like the dark side of the force: quicker, easier, and more seductive, but not the right way to go" [Hoc15]. Despite this, developers frequently begin their configuration management journey with the shell. It can be a familiar environment [Wei17], and is able to provide developers with a "quick and dirty" [Reda] solution to their configuration management needs. But, as many developers will eventually find, the shell is not without its growing pains [Hacc; Hacb; Hor17]. NASA ran head first into this problem when they needed to quickly migrate 65 legacy applications to AWS. After expanding to this new environment, they discovered that their existing process, based largely around an "ssh and run script" approach [DeK14], was not suitable to the new scale. This made it difficult to do seemingly simple tasks like patching servers and managing user accounts [Ans16].

Configuration management systems like Ansible are designed to overcome the challenges of scale and bring numerous benefits to developers. These systems can allow them to manage tens of thousands of servers simultaneously [Pupa], increase the frequency of their deployments [Rah18], reduce deployment time [Rah18] (sometimes from days to minutes [Pupa]), and slash costs [Rau18].

```
#!/bin/bash

curl "$HTTPS_CONFIG_TEMPLATES" -o /tmp/templates.tgz
tar zxf /tmp/templates.tgz
cp /tmp/templates/selinux.config /etc/selinux/config
setenforce 1
rm -f /tmp/templates.tgz
rm -rf /tmp/templates
```

Figure 5.1 A short shell script for enabling SELinux with a downloaded configuration file.

However, all these benefits come at a price: learning and migrating to a new configuration management system is challenging. The often recommended approach is to convert existing shell scripts one command at a time [Jay13; Redb; Raw18; Eas18], causing developers to both solicit and offer freelance services for doing so [Fre; Fiv]. Developers have even been known to ask for help converting individual commands, including `sed`, `apt-get`, `apt-key`, `curl`, and more [Redc; Redd; Stab; Staa; Stac]. One developer describes the learning process as "banging [your] head against solid objects," while another notes that they can feel their "eyes glaze over" every time they try to learn Chef [Haca; Pat13].

This work helps developers automatically push shell commands to Ansible module definitions. Our technique, named Dozer, uses system call tracing and a training set of open source playbooks to build a knowledge base of available Ansible modules. When asked to produce a migration for a shell command, Dozer uses the system call traces to match the shell command with similar Ansible modules. Then, Dozer uses the most similar modules as templates to generate and validate potential migrations of the shell command into an Ansible module. An implementation of our technique is available at `https://github.com/config-migration/dozer`.

This study evaluates Dozer on its ability to migrate common shell commands into Ansible modules. We find that it is successful for 38 out of the 62 commands used in our evaluation (61%). For example, when asked to migrate the command `rm`, Dozer correctly generates a migration to the Ansible module `file` with the correct path and `state: 'absent'`.

5.1 Motivation

Ayesha is a software developer and security expert working for a large organization with many production servers. Her team maintains a set of standardization shell scripts like the one seen in Figure 5.1 that ensure servers are configured with good security practices. However, as Ayesha and her team have scaled to more and more servers, they have increasingly encountered intermittent issues which have resulted in configuration errors.

For example, the script from Figure 5.1 is responsible for configuring system security settings, such as SELinux. The script downloads and unpacks a standard set of templates, copies the SELinux configuration file, enables SELinux enforcement, and then cleans up. Network blips

sometimes cause the `curl` command to fail, preventing `tar` and `cp` from unpacking and copying the configuration file to the correct location to be picked up by SELinux. To make matters worse, bash will, by default, continue to execute commands in a script and return the exit status of the last command. Because `rm -rf` always succeeds, Ayesha and her team cannot rely on exit status to determine when this has happened. They have decided to move to Ansible to overcome this and other challenges.

Ayesha is just learning Ansible and needs to produce an Ansible playbook which also enables SELinux with her team's standard configuration. This is proving to be difficult because there are more than 3k built in Ansible modules, and many of them have different names and parameters than their corresponding shell commands. In fact, Ansible does not have a module named `curl`, `tar`, `cp`, `setenforce`, or `rm`, although it does have similar modules with different names. This forces Ayesha to spend time searching for the appropriate Ansible module for each shell command and then figuring out how to use it to achieve the same effect.

She turns to Google, looking for specific examples to help set her on the right path. For example, she isn't sure how to remove a directory using Ansible like her bash script does. Fortunately, previous developers have published examples on how to use various Ansible modules, including this `file` module example.

```
1  file:
2    path:  '/etc/nginx/sites-enabled/default'
3    state: 'absent'
```

This example allows Ayesha to infer two things that she needs to migrate her uses of `rm`.

1. The Linux shell command `rm -rf` maps to the Ansible module `file` with the parameter `state: 'absent'`.

2. The `rm` filename maps to the `file` `path` parameter.

Using this and other examples, Ayesha eventually manages to discover the `get_url`, `unarchive`, `copy`, `selinux`, and `file` modules that allow her to perform the necessary configuration and uses them to create the resulting Ansible playbook shown in Figure 5.2.

Ayesha was able to synthesize migrations for each of the commands in her script because Ansible provides modules that perform the same actions. However, the process was not ideal. Not only did she have to spend time searching, since the modules do not share names with the corresponding shell commands, but after finding the correct module, Ayesha had to figure out what parameters to use to achieve the desired result. For example, Ayesha spent significant time debugging why her files were not copied on the server until she found that she was required to also add the parameter `remote_src: true`. Unfortunately, Ayesha still has to finish migrating the rest of her configuration scripts, and she wishes there was something that could help her.

```
hosts: all
tasks:
  - get_url:
      url:  "{{ HTTPS_CONFIG_TEMPLATES }}"
      dest: '/tmp/templates.tgz'
  - unarchive:
      src:  '/tmp/templates.tgz'
      dest: '/tmp/templates'
      remote_src: true
  - copy:
      src:  '/tmp/templates/selinux.config'
      dest: '/etc/selinux/config'
      remote_src: true
  - selinux:
      state:  'enforcing'
      policy: 'targeted'
  - file:
      path:  '/tmp/templates.tgz'
      state: 'absent'
  - file:
      path:  '/tmp/templates'
      state: 'absent'
```

Figure 5.2 An Ansible playbook performing the same actions as the shell script from Figure 5.1.

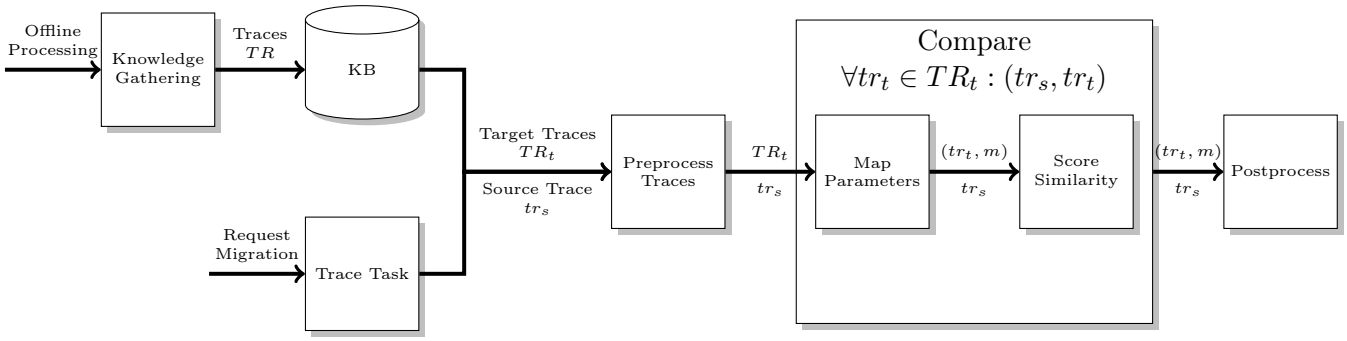


Figure 5.3 High level view of the steps in the Dozer knowledge gathering and migration process.

5.2 Dozer

Dozer is a technique and tool for pushing configuration task definitions between languages. It is based on the insight that a task, like a shell command or Ansible module, can only change the system state by communicating with the kernel via the system call interface. This shared interface provides an opportunity to observe how tasks interact with the system, and two or more tasks can be compared based on their interactions. The technique is comprised of two main parts: 1) offline knowledge gathering, where Dozer observes the execution of existing configuration scripts, and 2) migration, where Dozer compares configuration tasks and generates a migration. Figure 5.3 provides an illustration of the steps in the Dozer pipeline.

Figure 5.4 shows system calls made by `rm`. Each line notes the current process PID, the name

```
1  > rm filename.txt
2  23    unlinkat(-100, "filename.txt", 0) = 0
3  23    close(0</dev/pts/0<char 136:0>>)  = 0
4  23    close(1</dev/pts/0<char 136:0>>)  = 0
5  23    close(2</dev/pts/0<char 136:0>>)  = 0
6  23    exit_group(0)                     = ?
```

Figure 5.4 A subset of the system call trace produced by executing the `rm` command to remove a file named `filename.txt`. Some system calls are omitted for brevity.

of the system call, its arguments, and its return value. The core operation is the call to `unlinkat`, which instructs the kernel to remove the specified file. This action is followed by a call to `close` on file descriptors `0-2` (stdin, stdout, and stderr). Finally, the call to `exit_group` terminates the program with the success status code `0`.

Ansible's support for removing files is exposed through the `file` module, which makes several of the same system calls as `rm`, suggesting that finding and matching similar tasks across configuration languages is feasible. There is an important difference, though, since `file` uses the `unlink` system call instead of `unlinkat` to perform the deletion. The system calls may also have different arguments if different filenames are provided during tracing. To enable migration, we must overcome the challenges of identifying corresponding system calls with differing arguments and detecting the system calls which make up a task's "core" behavior.

5.2.1 Definitions

We refer to a sequence of system calls like this as a system call trace, or *strace* where not ambiguous. Likewise, we use *syscall* to refer to a system call. We refer to Ansible modules, Puppet resources, Docker `RUN` commands, and Linux shell commands as *configuration tasks* or *tasks*. Because Docker `RUN` commands are run through the shell, we treat them the same as Linux shell commands.

5.2.2 Knowledge Gathering

Knowledge gathering is the first stage of the Dozer process and is how Dozer learns about existing configuration tasks. This stage occurs offline, prior to a developer asking Dozer to generate a migration. In this stage, Dozer runs a configuration script such as an Ansible playbook in a Docker container and uses the `strace`[1] utility to collect a system call trace for each configuration task in the script. This allows us to efficiently gather traces for a wide variety of configuration tasks using existing open source scripts.

5.2.3 Generating Migrations

[1] `https://strace.io/`

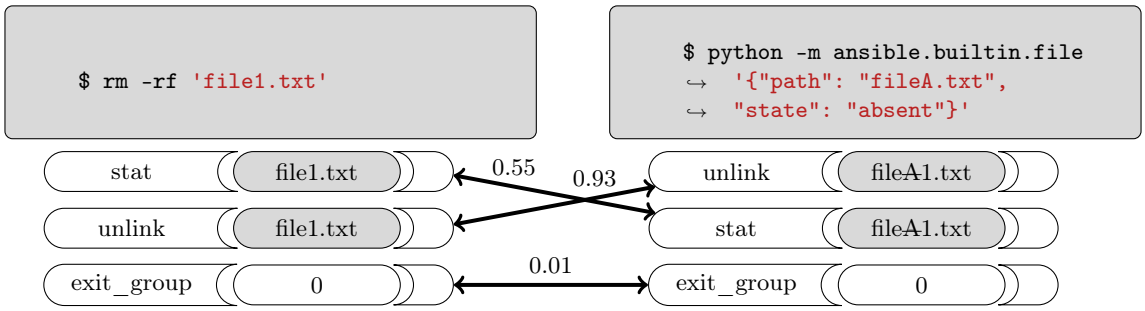


Figure 5.5 A high-level depiction of the strace comparison between the shell command rm and the Ansible module file. Dozer first searches for instances of parameters within syscalls, then determines the mapping that will result in the best score. Finally, it matches equivalent syscalls between the straces and assigns an overall comparison score based on the weighted scores of the matched syscalls.

Table 5.1 Strace comparison scores for an execution of `rm`.

Rank	Score	Task	Parameters	Map
		rm	`["-rf", "/var/lib/apt/lists/*"]`	
1	1.0	file	`{"path": "/etc/motd", "state": "absent"}`	`[1 ∼ path]`
2	0.9995	file	`{"dest": "/etc/ferm/rules.d/100_rule_avahi.conf", "state": "absent"}`	`[1 ∼ dest]`
3	0.9995	file	`{"path": "/etc/apt-cacher-ng/zz_debconf.conf", "state": "absent"}`	`[1 ∼ path]`
4	0.9368	file	`{"path": "/etc/nginx/sites-enabled/default", "state": "absent"}`	`[1 ∼ path]`
			⋮	
3578	0.0001	apt	`{"name": ["python3", ..., "python-dnspython"], "state": "present"}`	`[]`

[1] The top ranked match is a definition for the Ansible `file` module.
[2] Dozer has correctly mapped the first positional argument of `rm` to the `path` parameter (`dest` is an alias for `path`).
[3] Comparison scores for the same task may vary due to slight runtime differences in the straces.

Dozer searches for similar configuration tasks by comparing the strace tr_s of the source task to all target straces $tr_t \in TR_t$ from the knowledge base. Each target trace tr_t is given a *comparison score*, which is roughly based on the number of equivalent syscalls shared by tr_s and tr_t. A simple comparison scheme is to treat two straces as sets and compute their intersection-over-union, which provides some insight into how many shared vs distinct syscalls appear in each strace. This scheme performs well for long straces, but is sensitive to noise and cannot provide a mapping between the parameters for the two tasks. To overcome these challenges and improve Dozer's ability to perform correct migrations, we implement a suite of preprocessing techniques, an algorithm for mapping parameters, a method for normalizing syscalls, and a comparison scheme using information content.

Figure 5.5 gives a high-level view of how two straces are compared. All syscalls are first normalized to a standard representation, and then the input parameters `file1.txt` and `fileA.txt` are mapped together because this mapping has the highest overall influence on the comparison score. Finally, an aggregate comparison score is computed from the weighted scores for all matching pairs of syscalls between the two traces. Table 5.1 shows a portion of the comparison scores for the Linux shell command `rm` compared to Ansible modules from our evaluation.

The comparison procedure results in a set of similar tasks in the target configuration language and a parameter mapping from the source to target task. Dozer uses these results to generate and validate potential migrations and observes the effects of each migration on the system. It picks the migration which produces the most similar changes to the original source task.

5.2.3.1 Preprocessing

Preprocessors allow Dozer to reduce the impact of noise on the strace comparison process. One of the primary sources of noise is the current process PID, which is often included in temporary file names when generating lock files or when reading information from the proc filesystem (procfs). The PID is likely to vary across traces and can make syscalls appear different, even when they represent the same action. Dozer preprocesses syscalls which contain the current process PID, replacing them with a standard value. For regular filenames, that standard value is the string "PID." For access to procfs, `/proc/<pid>` is replaced with the equivalent expression `/proc/self`.

5.2.3.2 Mapping Task Parameters

Some tasks accept user provided parameters which are passed directly to syscall arguments. For example, the filename parameter in `rm` from Figure 5.4 is passed directly to the call to `unlinkat`. Before computing a similarity score for a pair of straces (tr_s, tr_t), Dozer attempts to find an optimal mapping of their task parameters. This mapping is a novel part of the Dozer process that aids in detecting similar tasks and is used when generating the resulting migration.

Dozer starts by instrumenting syscall arguments when either 1) a parameter is a file glob expression that could expand to the argument, or 2) a parameter is a substring of the argument.

Instrumentation allows Dozer to change the perceived value of the argument when a parameter mapping is applied during comparison.

We then find the set of all pairs of syscalls $A = \{(s_s, s_t) \mid s_s \in S_s \wedge s_t \in S_t \wedge \text{with\_map\_equal}(s_s = s_t) \wedge \text{with\_map\_distinct}(s_s \neq s_t)\}$, where S_s and S_t are the sets of all syscalls from the source and target straces, respectively. This set contains all pairs of syscalls that are equivalent when all task parameters are mapped to the same value, but are different when all task parameters are mapped to a distinct value, meaning that their equality can be affected by mapping source and target task parameters. We refer to them as the affected pairs.

The affected pairs are then used to compute the potential benefit of mapping two task parameters $p_s \sim p_t$ from the source task parameters P_s and target task parameters P_t, respectively. The benefit is computed in the following manner. Map $p_s \sim p_t$, and map all other parameters to the same value, thus creating two groups of "equivalent" parameters: $\{p_s, p_t\}$ and $(P_s \cup P_t) \setminus \{p_s, p_t\}$. The benefit$(p_s \sim p_t)$ is the sum of the portion of each affected pair from the set $A_m = \{(s_s, s_t) \in A \mid \text{with\_map}(s_s = s_t)\}$ that the mapping is responsible for. A mapping $p_s \sim p_t$ is responsible for syscalls $s_s = s_t$ if p_s is an argument of s_s (denoted $p_s \in s_s$) and p_t is an argument of s_t (denoted $p_t \in s_t$), and the portion is the number of occurrences of p_s and p_t divided by the total number of parameters in s_s and s_t.

The benefit is formally defined below as:

$$\text{benefit}(p_s \sim p_t) = \sum_{(s_s, s_t) \in A_m} \frac{\sum_{p_s \in s_s} \mathbf{1}_{\{p_s\}}(s_s) + \sum_{p_t \in s_t} \mathbf{1}_{\{p_t\}}(s_t)}{|s_s| + |s_t|}$$

The final parameter mapping is taken as the maximum matching with maximum weight of the graph $G = (V, E)$, where $V = P_s \cup P_t$ and $E = \{(p_s, p_t, \text{benefit}(p_s \sim p_t)) \mid p_s \in P_s \wedge p_t \in P_t \wedge \text{benefit}(p_s \sim p_t) > 0\}$.

5.2.3.3 Canonical Syscall Representation

We introduce a canonical syscall representation so that syscalls which produce the same effect on the system are given the same representation. This influences the overall comparison of straces, as configuration tasks may use different syscalls to perform the same actions. A canonical form may differ from the initial syscall definition for several reasons, which are outlined below.

5.2.3.3.1 Different Arguments

The Linux kernel provides multiple versions of some syscalls. These implementations have the same general behavior, but accept different arguments. In such cases, we convert the syscall to

a canonical form with a standardized name and arguments. Two such syscalls are `chown` and `fchown`, both of which can be used to change file ownership. `chown` accepts the file path as a string, while `fchown` accepts a file descriptor. We convert both syscalls to the canonical form `<CanonicalForm: (‘chown’, <file path>, <owner>, <group>)>`, resolving the full file path from the file descriptor in the case of `fchown`, and preserving the owner and group arguments.

5.2.3.3.2 Same Behavior

Some syscalls change their behavior based on the arguments that are provided to them. In such cases, we convert the syscall to a canonical form based on these arguments. One example is the syscall `unlinkat` which behaves like the syscall `rmdir` when provided the flag `AT_REMOVEDIR`, otherwise it behaves like `unlink`. We convert the syscall to the canonical form `<CanonicalForm (‘rmdir’, <path>)>` or `<CanonicalForm (‘unlink’, <path>)>` depending on the value of this flag.

5.2.3.3.3 Variable Arguments

Many syscalls accept arguments that do not affect the behavior of the syscall, such as memory addresses. In such cases, we convert the syscall to a canonical form with these arguments removed.

5.2.3.4 Comparing With Information Content

We introduce a comparison scheme which reduces the contribution to the comparison score caused by common syscalls matching. This is done by computing the information content [Sha48] of each syscall, defined as $\mathrm{ic}(s) = -\log P(s)$, where $P(s)$ is the probability of finding the syscall in a given strace from the set of all known straces. We then normalize the information content by dividing by the maximum possible value. This gives a high weight to rare events and a low weight to common events, intuitively giving us some measure of the “surprise” of finding a matching syscall in the source and target straces. We map all task parameters to an equal value here due to the complexity of finding an optimal mapping of task parameters for all straces.

Using the information content, we compute the comparison score as follows. For each syscall s which occurs in both the source strace tr_s and the target strace tr_t, count the number of times that s occurs in both tr_s and tr_t and take the minimum. We compute the total score as:

$$\text{nic}(s) = \frac{-\log P(s)}{-\log \frac{1}{|TR|}}$$

$$\text{count}(s) = \min(\sum_{s \in tr_s} 1_{\{s\}}(tr_s), \sum_{s \in tr_t} 1_{\{s\}}(tr_t))$$

$$\text{score}(s) = \frac{\sum_{s \in (tr_s \cup tr_t)} count(s) * \text{nic}(s)}{\frac{|tr_s|+|tr_t|}{2}}$$

We normalize by the average number of syscalls in tr_s and tr_t to lower the comparison score of straces with different lengths. This can happen when a long strace contains most of the syscalls made by a shorter strace, as happens for the Linux shell command `rm` and the Ansible module `user`. Despite this, `user` is a bad match for `rm` because it also makes a number of other syscalls that reflect its different behavior.

5.2.3.5 Generating and Validating Migrations

Finally, Dozer postprocesses the most similar target tasks to generate potential migrations for the source task. Potential migrations are validated, and the one that produces the most similar environment to the source task is taken as the final migration of the source task into the target configuration language. We now describe the algorithm for generating and validating migrations of a single target task in detail.

Dozer begins by applying the parameter mapping discovered during comparison. This produces a migration based on the target task that includes parameters from the source task. Dozer then executes the source task and migration against the same starting environments and observes the changes that each makes. Lastly, Dozer builds a tree structure from the migrated task's parameters and performs three postorder traversals, each focused on improving an aspect of the migration's parameters.

5.2.3.5.1 Preprocessing Migration Parameters

During the first postorder traversal, Dozer focuses on preprocessing the migration's parameters. At each node it visits, Dozer attempts replacing the node with a list containing the node, and accepts the change if the new migration produces exactly the same configuration changes as the old migration. Some configuration tasks, like Ansible's `apt` module, have parameters which will accept either a single value or a list of values. Doing this allows Dozer to convert any single-valued parameters to list-valued parameters, which is useful for improving the mapping.

5.2.3.5.2 Searching for a Better Mapping

During the second postorder traversal, Dozer focuses on improving the mapping of parameters from the source to target task. At each node it visits, Dozer picks the source parameters that have not yet been mapped to the target task, or which have been mapped to the current subtree and can be "pulled up," then generates a new migration with the parameter mapped to the current node. Dozer will also attempt extending lists by mapping parameters to the end of the list. If one or more new migrations result in configuration changes that are more similar to those of the source task than the old migration, Dozer selects the one that is the most similar.

5.2.3.5.3 Postprocessing Migration Parameters

During the third postorder traversal, Dozer focuses on removing unnecessary parameters. At each node it visits, Dozer removes the node and accepts the new migration if the configuration changes are more similar to those of the source task than the old migration.

5.3 Evaluation

We evaluate Dozer by its ability to accurately suggest migrations of common Docker `RUN` commands to Ansible modules. All source and target task definitions were automatically gathered from open source configuration scripts. We rate the quality of the suggestions by Dozer's ability to recommend the correct target task, recommend the correct parameters, and correctly map task parameters. We now describe each part of the evaluation in detail.

5.3.1 Datasets

We used two datasets to collect and trace configuration tasks for our evaluation. The first provides shell commands, and the second provides Ansible modules. All tracing was performed in a Docker container based on Debian Buster using `strace` version `4.26-0.2`. For shell commands, we ran the tracing process directly inside of a clean container. Ansible modules used Python version `3.8` and Ansible version `2.9.2`. We stopped tracing long-running tasks after 50K syscalls due to the performance overhead of tracing and because this is usually sufficient for comparison.

5.3.1.1 Dockerfiles

Our source tasks are Linux shell commands used as `RUN` commands in open source Dockerfiles on GitHub. We began by using Google BigQuery to collect the names of all repositories containing a single stage Dockerfile based on the Debian or Ubuntu images. This query returned 11,787 unique repositories.

We then cloned each Dockerfile and extracted each `RUN` command. `RUN` commands that were in the Dockerfile exec syntax were converted to shell syntax by joining each part with a space separator. Then we split the command into one or more shell commands separated by the

characters `;`, `|`, or `&`. We consider the first positional argument of the task definition to be the executable name, and the remainder to be the arguments. This generated 35,040 unique task definitions with 83,695 invocations.

Finally, we selected the 100 most commonly invoked tasks for use in the experiment, then filtered out definitions where the task:

- Executable is not valid. This can occur when variables such as `DEBIAN_FRONTEND = noninteractive` occur as the first positional argument instead of the executable name.
- Is an interpreter (like `python`), build system (like `make`), or script (like `./configure`), as these perform many different changes based on an external script.
- Requires interactive user input, due to difficulty with automated tracing.
- Is a shell builtin like `set`.
- Does not modify the system configuration, such as `true`.

After filtering, 62 Linux tasks remained for use in the experiment. We provided a short setup script for tasks that required a starting state for the environment. For example, for tasks that used the executable `curl`, we provided the setup script `apt-get install -y curl`. This is similar to what a developer might do if they wanted to trace a shell command manually. We then traced all tasks in a clean Docker container based on Debian Buster.

5.3.1.2 DebOps

We built our knowledge base of available target tasks from version 1.2.0 of the DebOps suite.[2] DebOps is a management suite for Debian based systems that relies on Ansible, and it provides a collection of Ansible roles and playbooks for configuration. The main playbook, `site.yml`, configures user accounts, authentication, networking, services, and more.

To trace Ansible modules in the DebOps suite, we generated an Ansible inventory file that places localhost as a member of all host groups, ensuring that all roles and tasks will be run against it. We then executed the playbook `site.yml` against a clean Docker image based on Debian Buster with Python 3.8. Ansible execution is performed programmatically, allowing us to intercept module invocations and trace their execution. Tracing the DebOps playbooks yielded 3,578 straces for 26 Ansible modules with 1,296 unique invocations.

5.3.2 Methodology

We ran our evaluation by asking Dozer to migrate the 62 Linux tasks gathered in Section 5.3.1.1 to Ansible modules. We then inspected each migration and rated it based on Dozer's ability to select the correct Ansible module, select the correct module parameters, and correctly map

[2] `https://docs.debops.org/en/stable-1.2/`

shell command parameters to Ansible module parameters if applicable. We consider a successful migration to be one where Dozer suggests the correct module with a set of parameters such that, should they be mapped to the source task parameters, the module would produce the correct effect on the system.

5.3.3 Experiment Hardware

The experiment was run on Ubuntu 20.04 LTS virtual machines configured with 4 CPUs and 40GB memory. The VMs were run on a VMware ESXi 6.5.0 hypervisor backed by a Dell PowerEdge R640 server with an Intel Xeon Silver 4116 CPU at 2.10 GHz.

5.4 Results

Overall, Dozer successfully migrated 38 out of 62 commands from our evaluation (61%) and generated complete parameter mappings for 33 out of the 38 successful migrations (87%). Table 5.2 shows migrations generated during our evaluation. There are two categories of failed migrations. The first is when Dozer is unsuccessful at finding and migrating to the correct Ansible module definition even though a similar module exists in its knowledge base, which we discuss in Section 5.4.2. This was the case for 7 commands. Dozer also failed to generate a correct migration for commands that are unsupported by Ansible or commands whose corresponding modules did not appear in the knowledge base, which was the case for 7 and 10 commands, respectively.

5.4.1 Successful Migrations

Dozer is able to migrate most Linux tasks if a corresponding Ansible module appears in its knowledge base. Because Dozer observes syscalls, Linux and Ansible tasks do not need to have similar names, nor do their recorded definitions need to use the same parameters.

5.4.1.1 The Migration Process

Dozer is able to correctly migrate file removals using the shell command `rm` to file removals using the Ansible module `file`, as seen in Table 5.2. Here is how each step of the pipeline operates.

5.4.1.1.1 Straces

Dozer traces `rm` and loads all recorded straces for Ansible modules. Among these is the strace for the similar `file` module in Table 5.2, which we will focus on. The `rm` strace contains 34 syscalls, the key call being `unlinkat(-100, "/bin/sh", 0)`. The `file` strace contains 2101 syscalls, the key call being `unlink("/etc/nginx/sites-enabled/default")`.

Table 5.2 Sample migration results from the Dozer evaluation. *Shell Commands* are matched to *Similar Modules* from the Dozer knowledge base, which are then used as templates to generate final *Migrations*.

Shell Command	Similar Module	Migration	Successful

`rm '/bin/sh'`	`file:`
	`  path:  '/etc/nginx/sites-enabled/default'`
	`  state: 'absent'`
		`file:`
		`  path:  '/bin/sh'`
		`  state: 'absent'`
			✓

`echo 'daemon off;' >> /etc/nginx/nginx.conf`	`lineinfile:`
	`  dest:  '/root/.profile'`
	`  regexp: '^.*mesg n.*$'`
	`  line:  'tty -s && mesg n || true'`
	`  state:  'present'`
		`lineinfile:`
		`  dest:  '/etc/nginx/nginx.conf'`
		`  regexp: '^.*mesg n.*$'`
		`  line:  'daemon off;'`
		`  state:  'present'`
			✓

`locale-gen 'en_US.UTF-8'`	`locale_gen:`
	`  name:  'en_US.UTF-8'`
	`  state: 'present'`
		`locale_gen:`
		`  name:  'en_US.UTF-8'`
		`  state: 'present'`
			✓

`apt update`	`apt:` `  update_cache: true`	`apt:` `  update_cache: true`	✓

`apt-get install -y git`	`apt:`
	`  name:  ['git', 'less', 'ncurses-term', 'tmux', 'yadm']`
	`  state: 'present'`
		`apt:`
		`  name:  'git'`
		`  state: 'present'`
			✓

`dpkg-divert --local --rename --add /sbin/inictl`	`dpkg_divert:`
	`  path:  '/etc/crontab'`
	`  state:  'present'`
	`  delete: true`
		`dpkg_divert:`
		`  path:  '/sbin/inictl'`
		`  state:  'present'`
		`  delete: true`
			✓

`ln -s /app /var/www/html`	`file:`
	`  path:  '/var/www/html'`
	`  state: 'directory'`
		`file:`
		`  path:  '/var/www/html'`
		`  state: 'directory'`
			✗

`mkdir -p /var/run/sshd`	`lineinfile:`
	`  dest:  '/etc/at.allow'`
	`  regexp: '^root$'`
	`  line:  'root'`
	`  state:  'present'`
	`  create: true`
	`  owner:  'root'`
	`  group:  'daemon'`
	`  mode:  '0640'`
		`lineinfile:`
		`  dest:  '/var/run/sshd'`
		`  regexp: ['^root$']`
		`  line:  ['root']`
		`  state:  'present'`
		`  create: true`
		`  owner:  'root'`
		`  group:  'daemon'`
		`  mode:  '0640'`
			✗

`mv composer.phar /usr/local/bin/composer`	`copy:`
	`  src:  '/root/.ansible/tmp/ansible-tmp-1583436619.4685626-258219439746011/source'`
	`  dest:  '/etc/ansible/facts.d/root.fact'`
	`  mode:  '0755'`
	`  follow:  false`
	`  checksum: '...'`
		`copy:`
		`  src:  'composer.phar'`
		`  dest:  '/usr/local/bin/composer'`
		`  mode:  '0755'`
		`  follow: false`
			✗

5.4.1.1.2 Preprocessing

Both straces are preprocessed according to Section 5.2.3.1. Neither of the key unlink syscalls are affected.

5.4.1.1.3 Parameter Mapping

The parameter `/bin/sh` is found in 3 `rm` syscalls. Likewise, `/etc/nginx/sites-enabled/default` is found in 6 `file` syscalls. Using the procedure outlined in Section 5.2.3.2, Dozer determines that considering these parameters to be the same results in the largest number of syscalls from the two straces matching. This mapping is used for comparison.

5.4.1.1.4 Scoring

A score is computed using the weighted scheme described in Section 5.2.3.4. During comparison, both unlink syscalls are converted to the canonical form `<CanonicalForm ('unlink', <Parameter>)>` and are considered equal. Calling unlink on a parameter is a relatively unique operation, and our comparison scheme assigns this a high enough score to outweigh the more common syscalls, causing Dozer to consider the straces similar.

5.4.1.1.5 Postprocessing

Finally, Dozer uses the similar module and mapping to generate and validate the migration from Table 5.2 according to Section 5.2.3.5. The source shell command and migrated Ansible module produce the same effect on the system.

5.4.1.2 Complex Interactions

Migrations can even be produced for tasks that interact with the system through complex mechanisms like shell redirection. Table 5.2 shows the results for `echo`, which would not normally produce a change to the system. However, its stdout has been redirected, causing the setting `daemon off;` to be appended to the nginx configuration file.

Even though `echo` does not open the configuration file itself, the file write appears in the command's strace and is observed by Dozer. Dozer then compares the `echo` strace against straces for Ansible modules, and determines that a subset of them are similar enough to the shell command to warrant further investigation. Most of these similar modules interact with the filesystem in some way, and both `blockinfile` and `lineinfile` are among the candidates. Both can be used to ensure a file's contents, which is exactly the effect that Dozer is trying to reproduce.

One of the specific instances of `lineinfile` is the similar module shown in in Table 5.2. Dozer inferred the mapping `[2 ∼ dest]` during comparison. This is close, but missing the additional mapping `[0 ∼ line]`. Dozer generates and validates migrations based on the similar module

and its initially inferred mapping as described in Section 5.2.3.5. During postprocessing, Dozer learns that the additional mapping `[0 ∼ line]` makes the effects of the migration more similar to those of the `echo` command that it is trying to reproduce, and so it accepts the additional mapping. The final migration, shown in Table 5.2, uses the `lineinfile` module to append the `daemon off;` setting to the nginx configuration file, exactly the same as the original shell command.[3]

5.4.1.3 Complex Tasks

Migrations are also produced for complex tasks. The locales package used on Debian systems does not include localization files due to the prohibitive size of the files. Instead, the package provides the `locale-gen` command for generating specific localization files on demand. Dozer correctly migrates the command `locale-gen 'en_US.UTF-8'` as shown in Table 5.2.

5.4.1.4 Same Module, Different Parameters

Dozer handles different commands and subcommands that map to the same Ansible module. Table 5.2 shows how the `apt` and `apt-get` commands from both migrate to the Ansible `apt` module with different parameters. Dozer is able to do this by using different modules from the knowledge base as templates for migration.

5.4.1.5 Custom Modules

Our technique is not limited to a pre-defined set of tasks from any configuration language, and Dozer is able to learn about new modules simply by tracing them. The DebOps dataset described in Section 5.3.1.2 adds at least one such custom Ansible module, named `dpkg_divert`.[4] During our evaluation, Dozer migrated `dpkg-divert` to this module. This migration shows how Dozer can easily incorporate information about new tasks without requiring any developer intervention, a useful ability as configuration languages evolve.

5.4.2 Unsuccessful Migrations

Dozer was unsuccessful at producing a migration for 7 of the Linux tasks in our evaluation. Unsuccessful migrations fall into one of two categories: 1) Dozer suggests a migration to the correct module, but the suggested parameters are incorrect, or 2) Dozer suggests a migration to the wrong module.

[3] Note that the `lineinfile` module appends by default if `regexp` is not matched. The parameter does not get removed during parameter postprocessing because doing so does not improve the migration under the conditions seen in the evaluation.

[4] `https://github.com/debops/debops/blob/stable-1.2/ansible/roles/debops.ansible_plugins/library/dpkg_divert.py`

5.4.2.1 Correct Module, Incorrect Parameters

The shell command `ln` is an example of where Dozer produces an incorrect migration to the correct module. Here, Dozer correctly migrates to the `file` module, but it does so with the parameter `state: 'directory'` instead of the expected `state: 'link'`. Unsuccessful migrations like this can occur if the correct definition is not ranked high enough in the comparison scores, or if the effect produced by the module is not detected by validation during postprocessing.

5.4.2.2 Incorrect Module

In some cases, unsuccessful migrations resulted from Dozer being unable to detect similar behavior. This can happen when tasks operate in ways that make their behavior hard to observe. The `mkdir` command is a good example of this. The command normally accepts a directory name as a parameter and passes it to the `mkdir` syscall. This behavior can be matched with the Ansible `file` module, which also passes its file parameter to the `mkdir` syscall with `state: 'directory'`. When called with the `-p` flag, `mkdir` behaves differently, as seen below.

```
1  > mkdir -p /var/run/sshd
2  <CanonicalForm ('mkdir', '/var', 511)>
3  <CanonicalForm ('chdir', '/var')>
4  <CanonicalForm ('mkdir', 'run', 511)>
5  <CanonicalForm ('chdir', 'run')>
6  <CanonicalForm ('mkdir', 'sshd', 511)>
```

Rather than passing the entire parameter, `mkdir` splits the path into components, then continues to create and move into each component. This prevents Dozer from matching its behavior with the Ansible `file` module.

5.4.3 Unsupported or Unknown Behavior

A perfect migration does not always exist. For 10 shell commands, we determined that Dozer did not have an appropriate module definition in its knowledge base to use for migration. Because Dozer relies on observing and matching behavior, it was unable to correctly migrate them. For example, our Docker data set contained the task `gem install bundler`, a dependency installation task specific to the Ruby programming language. While a corresponding `gem` module exists in Ansible, none of the playbooks in our dataset made use of it. Low coverage can be overcome by increasing the number and sources of configuration scripts to ensure a greater chance of observing all possible module behaviors.

In other cases, a migration did not exist because Ansible itself does not support the necessary behavior. Even when a task's behavior is not fully supported, Dozer was sometimes able to suggest migrations that produced a similar effect. For example, Ansible does not support moving files because doing so is not idempotent. It may be sufficient, and is often recommended, to use the `copy` module to copy the file to the desired location. When asked to migrate the shell

command `mv`, Dozer determines the `copy` module produces the most similar effect and migrates to it.

5.5 Limitations

Dozer presents a novel approach to migrating configuration tasks. While we show that it is capable of migrating a large number of tasks correctly, there are some limiting factors that make it challenging to correctly migrate all tasks.

5.5.1 Assumption of a 1:1 Migration

Dozer's focus is to provide migrations at the individual task level. This approach has an implicit assumption that there is a 1:1 correspondence between tasks in different configuration languages. This is frequently the case for many common units of work found in configuration scripts. However, there are some cases where a single task in one configuration language might represent more than one task in another. Take the following task definition for the Ansible module `make`.

```
1  make:
2    chdir:  '/app'
3    target: 'default'
```

The task definition provides the parameter `chdir`, making it equivalent to `cd /app && make default`. Dozer currently does not consider such combinations, but could be extended to by looking at groups of straces.

5.5.2 Knowledge Base Generation

Our technique learns about existing configuration tasks and their behaviors by observing their execution. While this grants Dozer the ability to easily incorporate knowledge about new configuration tasks, it introduces the fundamental limitation that we must observe a task being used before being able to suggest migrations to it. Further, as discussed in Section 5.4.3, there are instances where tasks do not match because Dozer has not observed all possible behavior for the task.

5.6 Discussion and Future Work

Our results show that Dozer is able to successfully migrate 38 out of 62 common Linux shell commands to Ansible modules (61%). Dozer also searches for a mapping from source to target task parameters, which were successfully generated in 87% of cases. Migrations are made after simply observing the execution of existing configuration scripts. Dozer also provides developers with a method for verifying that migrations produce the intended configuration change on the filesystem, which can help avoid the unintended effects of incorrect configuration changes [Sev14;

Git]. This represents an important step towards automated configuration language migration. There is, however, room for improvement that will lead to better migrations.

5.6.1 Process Metadata

Dozer currently records very little information about the processes that carry out configuration tasks. In addition to the task definition, the task's strace indicates the current process identifier (PID). Information about the PID is used to normalize syscalls according to some very simple rules. No other information is recorded or used during Dozer's comparison implementation. Lack of information can impede Dozer's ability to correctly compare syscalls, and we might achieve more accurate suggestions by combining more recorded process metadata with additional preprocessing rules.

Two syscalls that refer to the same file by absolute and relative paths, respectively, will not compare equal even if they do represent the same action on the system. We could address this by recording the current working directory when the task is executed, plus any directory changes made during execution. Then, during preprocessing, we could use this information to convert all relative file paths to absolute file paths, allowing for direct comparison. Doing this would be a step in the right direction towards fixing the path detection for `mkdir` as seen in Section 5.4.2.

We may also be able to better handle the occurrence of PID in file paths by recording information about the task process and its ancestors or descendants while running, allowing us to create better preprocessing rules or find an optimal mapping of PIDs from two straces, similar to how we handle task parameters.

5.6.2 Reduce Noise

Short straces can be subject to noise which affects how they compare to other straces. A typical strace for `mkdir` contains about 80 syscalls (it may vary due to different code paths taken during execution). The command's core behavior is represented by a single `mkdir` syscall. The remaining 79, related to loading libraries and memory management, are not directly related to the system changes being made. While our weighting scheme described in Section 5.2.3.4 is frequently able to correctly emphasize important syscalls, if two tasks happen to both load relatively uncommon libraries, the related syscalls can dominate the comparison.

Future work can investigate how to best handle these situations. It may be sufficient to only consider syscalls which represent a change to the system configuration, such as `mkdir`, and syscalls which represent a change to process execution with respect to the configuration environment, such as `chdir`.

5.6.3 Combining Tasks

As we noted in Section 5.5.1, Dozer currently assumes that there is a 1:1 migration between tasks in different configuration languages. This is not necessarily the case. We may be able to

handle more complex migrations in the future by considering combinations of straces in the target configuration language which, considered together, exhibit behaviors which match those of the source configuration task.

5.6.4 Validation

The validation technique used to generate and validate migrations may be sensitive to noise and discount valid migrations. For example, the Ansible `user` module creates a home directory with configuration files by default. This behavior differs from the `useradd` command, which might not create a home directory depending on `/etc/login.defs`. This behavior might not matter to the developer whose goal is simply to add a new user. Other commands may make changes that validation currently does not detect, such as to file ownership and permissions, or might affect an entirely different system (such as `ssh`). We can overcome these challenges by introducing new validation techniques that consider additional properties and effects made to remote systems.

5.6.5 Full Script Migration

Migrating individual tasks is a critical step on the path to being able to migrate entire configuration scripts, since different configuration languages often have tasks in their domain specific languages which represent the same unit of work. Additional challenges are expected to appear with full-script migration, including behavior that exists outside of the task definition like registering task output as use for an input value later.

5.7 Related Work

Configuration is a challenging task that continues to occupy developer time and effort [Hor19a; McI11; Seo14; Sul16; Url18; Hor18]. Weiss, Guha, and Brun focus on repairing Puppet manifests based on commands executed in a Linux shell environment [Wei17]. Their technique, named Tortoise, models Puppet manifests with the ΔP modeling language. They then observe developer actions in the shell, using the effected changes as constraints on the model to synthesize a patched version. While Tortoise is similar to Dozer in its ability to translate information between configuration languages, it differs in a few crucial respects. Primarily, it is focused on synthesizing patches of existing cross-language configuration scripts, where Dozer focuses on generating complete task definitions without an existing script. In addition, ΔP requires manually modeling the effects of all configuration tasks and syscalls and focuses on the filesystem, whereas Dozer does not require a developer defined model, allowing it to easily incorporate new tasks.

Other recent work on configuration scripts has also focused on repair. Macho et al. present BuildMedic for dependency related faults in Maven builds [Mac18]. Hassan and Wang present HireBuild, which repairs gradle scripts based on history [Has18]. Importantly, both techniques operate within the same configuration language, instead of porting configurations or repairs across configuration languages. Horton and Parnin focus on inferring environment configurations

and synthesizing configuration scripts, but they only focus on application dependencies [Hor19a; Hor19b].

Su et al. implement HitoshiIO, which observes the behavior of programs running on the JVM and records an *execution profile* which can be used to find functionally similar code [Fan16]. Like HitoshiIO, we also use an execution profile, but perform comparison on Linux system calls. We additionally implement several preprocessing routines, provide a comparison scheme which discounts common actions, and demonstrate how common syscalls can be used to map inputs between tasks, allowing for task-migration.

In the migration domain, existing work focuses on general purpose programming languages. Xu et al. focus on producing migrations from older versions of an API to newer versions of the same API [Xu19]. Chen focuses on finding similar APIs in different libraries, but focuses on only a single programming language [Che20a]. Zhong et al. and Nguyen et al. go farther, focusing on mapping APIs between different programming languages [Zho10; Ngu16]. However, Zhong et al. require having the same project implemented in two different programming languages, which introduces a significant burden in mining API mappings. Xu et al. are able to take advantage of program structure to find similar APIs, but that structure may not be as clear in configuration languages. Crucially, no technique considers the underlying library or system calls made by the APIs as a source of generating mappings for migrations.

5.8 Conclusion

We present Dozer, a technique to help developers push their shell commands into higher level configuration languages. Dozer does not require any domain knowledge of the source or target language, and generates migrations by observing the execution of tasks from existing open source configuration scripts. Dozer is based on the key insight that configuration tasks that produce the same effects on a system are likely to make similar system calls. By recording the system calls made during task execution and providing a standard method for comparing them, we can detect and migrate between similar tasks. Dozer successfully migrated 38 common shell commands to Ansible modules, including non-standard modules. In addition, it was able to correctly map parameters for many of these tasks.

CHAPTER

6

SYNTHESIZING CONFIGURATION SCRIPTS

Currently under review

6.1 Introduction

Most programs cannot be executed successfully without a corresponding environment configuration that provides necessary external dependencies, configuration files, and other resources [Seo14; Yan16; Hor18]. Unfortunately, configuration management is a challenging and error-prone task for developers, even when using modern configuration management systems like Docker [Hor18; Hen21]. The challenge is due in part to the sheer number and scope of possible system configurations combined with the difficulty of discovering where configuration resources should come from and how they should be applied to the system.

This challenge exposes a clear need for tools that can help developers write configuration scripts. However, recent research has either focused on repairing/patching existing configuration scripts [Wei17; Hen21; Mac18; Has18], generating configuration scripts for limited scopes such as dependencies [Hor18; Hor19a; Hor19b], or synthesizing only single units of configuration scripts [Hor22a]. More research is needed to cross the gap from tools that help developers make small repairs to existing environments into tools that help developers synthesize full environments that enable program executability.

We address this gap with Synth, a novel technique for synthesizing full configuration scripts. Our synthesis technique is primarily based on the observation that a configured image is the result of a base (unconfigured) image plus a set of configuration changes. A configuration script can be synthesized by finding the ordered set of configuration tasks (such as shell commands or Ansible modules) that fully cover the desired set of configuration changes. Synth does not require an existing configuration script to build from and does not require input from the user during synthesis. Synth's core synthesis implementation is language agnostic, and it currently supports writing shell scripts, Dockerfiles, and Ansible playbooks.

To enable tractable performance, we trained Synth using a human-in-the-loop approach inspired by Shipwright [Hen21]. During training, Synth parses configuration tasks from a training set, clusters them by name, and then presents each cluster to a practitioner. The practitioner then either selects a representative configuration task from the list or writes a new configuration task that is representative of the intended behavior. For example, rather than training on every execution of `apt-get install <package>` from our training set, we defined a representative instance that would provide the baseline execution profile and configuration changes. Synth then executes the representative examples, diffs the original and final environments to extract a set of configuration changes, and records the execution and changes as a template in its knowledge base. As long as the command execution is deterministic, Synth can match the recorded templates against all executions of the configuration task. During use, Synth accepts a set of desired configuration changes plus a target configuration language, uses its recorded configuration templates to determine which configuration tasks can cover the desired changes, and then returns an executable configuration script to the user. An implementation of Synth is available at `https://github.com/config-synthesis/synth`.

We evaluated Synth using a dataset of 398 open-source Dockerfiles based on the official Debian and Ubuntu Docker images. For each Dockerfile in the test set, we extracted the configuration changes made to the base image, provided them to Synth as a set of desired changes to reproduce, and asked Synth to synthesize a new Dockerfile. Synthesized Dockerfiles were graded on their similarity to the original image in two ways: 1) Does the synthesized environment configuration have a similar or equivalent change set to the original? 2) Is the default `ENTRYPOINT` or `CMD` execution result the same in the synthesized image as it is in the original image?

Synthesis returned 100 executable Dockerfiles under a 30 minute timeout. The synthesized Dockerfiles enabled executability for 45/100 (45 %) of the images while the remaining configuration scripts provided a starting configuration that can be modified by developers. We compared the synthesized change sets to the desired change sets using the Jaccard coefficient (intersection over union) and found that synthesized images had an average of 0.9558, indicating that the synthesized change sets are highly similar to the desired ones. We also found that the synthesized configuration scripts correctly use high-level commands like `adduser` or `apt-get install` to create desired changes.

6.2 Synth

Our configuration script synthesis technique, Synth, is based on the idea that a valid configuration script should reproduce a set of desired changes when applied to a base environment. Configuration changes are produced by some configuration task like a shell command, Puppet resource, or Ansible module. If we know about available configuration tasks and the changes they make, then the job of configuration script synthesis simplifies to two steps:

1. Resolve the set of all configuration tasks that can be used to produce the desired changes.
2. Resolve the correct ordering for running tasks in the configuration script, including any additional commands needed to resolve errors.

The first step is highly related to the theoretical problem Set Cover, where the goal is to find a minimal covering for a desired target set. We first describe the theoretical approach to Synth based on this observation. We then discuss how Synth represents and records configuration tasks. Finally, we build on these ideas to describe how Synth is able to determine which configuration tasks are needed to produce a set of configuration changes and address the problem of ordering configuration tasks to produce an executable configuration script.

6.2.1 Configuration as an Instance of Set Cover

Set Cover is a well-known NP-Complete problem [Kar72]. In the decision variant of Set Cover, one is given a set U of elements known as the universe, a family S of subsets of U where the union of all sets in S is U, and an integer k. The goal is to determine if a covering C exists where each element of C is a subset from S, the size of C is at most k, and the union of all elements in C is U. The optimization variant of Set Cover does not provide an integer k, but instead asks for the covering C with minimum size. We formally define it as:

Set Cover (Optimization)

Input:	$U = \{e_1, e_2, \ldots, e_n\}$
	$S = \{s_1, s_2, \ldots, s_m \mid U = \cup_1^m s_i\}$
Problem:	Find a covering $C \subseteq S$ such that $U = \cup C$.
Objectives:	Minimize $\lvert C \rvert$.

We define an analogous optimization problem for configuration synthesis which we call Configuration Cover. In Configuration Cover, one is provided with a set of desired configuration changes U (the universe) and it is assumed that one knows about a family of configuration tasks T whose configuration changes union to U. We further take a weighted approach with Configuration Cover, where the goal is to find a covering C that completely covers U with the minimum amount of "excess" configuration changes. The weighted formulation becomes useful

when the set of known configuration tasks is not a perfect match for the actual set of changes, which can happen due to having an incomplete knowledge base of configuration tasks, differences between changes made by different configuration systems, noise, nondeterminism, and more. We formally define Configuration Cover as:

CONFIGURATION COVER

Input:	$U = \{c_1, c_2, \ldots, c_n\}$
	$T = \{t_1, t_2, \ldots, t_m \mid U \subseteq \cup_1^m t_i\}$
Problem:	Find a covering $C \subseteq T$ such that $U \subseteq \cup C$.
Objectives:	Minimize $\frac{\lvert\cup C\rvert}{\lvert U\rvert}$.

The solution to Configuration Cover (Section 6.2.3) will tell Synth which configuration tasks need to be in the final configuration script. However, these tasks must additionally be ordered such that they can execute correctly (Section 6.2.4). For example, `apt-get update` usually must come before `apt-get install`.

6.2.2 The Synth Knowledge Base

The Synth data model defines classes that represent configuration tasks, configuration changes, configuration errors, and argument mappings. The first two allow Synth to represent an invocation of a configuration task and the changes it makes to the system. The third records error information when a task fails. The last can be used to apply transformations to the other three, an operation that lets Synth use a single configuration task record as a template for changes made by invoking the same task with different arguments. Synth's data model can be used to represent any change that can be learned by diffing the system state from before and after a configuration task is run. We primarily define change types that deal with the filesystem, as these are the most common changes made by configuration tasks, but are not limited to them. Figure 6.1 shows the data model for the shell command `systemctl start nginx`.

Synth maintains a knowledge base that contains information about configuration tasks. The knowledge base is built automatically by running configuration scripts and recording each configuration task and the changes it makes. During analysis, Synth also speculatively executes configuration tasks that occur later in the script against a copy of the current environment. The first and last appearance of any errors are recorded, and the configuration tasks run between those two points are considered to be the resolving tasks for that error. For example, the `systemctl` task from Figure 6.1a will encounter the error in Figure 6.1b if `nginx` is not installed. Synth records that this is resolved by running `apt-get update && apt-get install -y nginx`.

```
ConfigurationTask:
  system = 'shell'
  executable = 'systemctl'
  arguments = ['start',  'nginx']
  changes = {
    ServiceStart:
      name = 'nginx.service'
  }
```

(a) systemctl start nginx.

```
ShellTaskError:
  exit_code: 5
  stdout: 'Failed to start nginx.service:
  ↪  Unit nginx.service not found.'
```

(b) An error encountered by systemctl start when the systemd service unit is not defined.

```
ConfigurationTask:
  system = 'shell'
  executable = 'systemctl'
  arguments = ['start', 'mysql']
  changes = {
    ServiceStart:
      name = 'mysql.service'
  }
```

(c) systemctl start mysql.

```
ArgumentMapping:
  'nginx' => 'mysql'
```

(d) An argument mapping that translates the arguments from Figure 6.1a to those in Figure 6.1c.

Figure 6.1 An example of the Synth data model. Synth preserves configurations tasks and their changes. The data model can also specify task errors and mappings that transform one task into another.

6.2.3 Inferring Configuration Sets

The first step in synthesizing a configuration script is determining which configuration tasks are needed to produce the correct configuration changes. This is the instance of our Configuration Cover problem defined in Section 6.2.1. Synth searches for an unordered set first, deferring the job of ordering until the entire set is known.

Although Configuration Cover is closely related to Set Cover, there are challenges that make an exact solution infeasible. In general, changes may be affected by external state such as network availability or be affected by other sources of nondeterminism that make comparison hard or impossible. Additionally, Synth may not know how to reproduce a configuration change if the appropriate task is not in its knowledge base.

Due to these challenges, we implement a greedy algorithm for solving Configuration Cover that performs well in practice. Our algorithm continually selects the maximum weighted configuration task from the knowledge base and adds it to the cover while there are still uncovered changes. The weight for each task is determined by how many of its changes intersect with the set of uncovered changes. Any changes that cannot be covered by the configuration tasks in Synth's knowledge base are discarded. Algorithm 6 outlines the algorithm in detail.

Synth does not use strict equality for intersecting a known configuration task's changes with the set of desired changes. This is due to the fact that the configuration task definitions Synth knows about likely use different arguments, and therefore have different effects on the system, than those in the target set. Using strict equality would mean that Synth needs to have an example of every possible configuration change in its knowledge base to guarantee successful

```
Procedure ConfigurationCover(system, desired_changes)
    system_tasks = get_applicable_tasks(system, changes)
    cover = set()
    while desired_changes do
        max_weight = 0
        max_weight_task = null
        for task in system_tasks do
            intersection = change_intersection( task.changes, desired_changes)
            weight = length(intersection) / length(task.changes)
            if weight == 0 then
                system_tasks.remove(task)
            end
            if weight > max_weight then
                max_weight = weight
                max_weight_task = task
            end
        end
        if max_weight_task then
            covering.add(max_weight_task)
            desired_changes -= max_weight_task.changes
        end
        else
            break
        end
    end
    return cover
```

Algorithm 6: Synth's configuration cover algorithm for finding a set of covering configuration tasks for a target configuration system like Docker or Ansible.

synthesis.

Instead, we use the configuration task definitions in Synth's knowledge base as templates that can be matched to another invocation of the same task with different arguments. We do so by starting with the observation that most executables used for configuring a system do so deterministically. When their outputs differ, it is usually because of one or more user provided arguments. For example, consider the configuration tasks from Figure 6.1. Both are invocations of `systemctl start <service>`, but their resulting changes have different values based on their service name. If we apply the mapping from Figure 6.1d to `systemctl start nginx`, the resulting task definition and changes will be equivalent to those produced by `systemctl start mysql`.

Our change intersection algorithm searches for the mapping which results in the largest possible intersection between a known configuration task's changes and the changes in the target set. Possible mappings are inferred by attempting to align two changes. For example, assume that Synth contains the configuration task for `systemctl start nginx` from Figure 6.1a in its knowledge base and that the configuration change for starting MySQL from Figure 6.1c has been provided as a desired change. Synth will align the changes as follows and infer that the changes will intersect if the service name `nginx` is replaced with the service name `mysql`.

```
<nginx>.service
   |   ||||||||          nginx => mysql
 mysql .service
```

The inferred mapping is used to transform the template task from Figure 6.1a into the final task definition from Figure 6.1c that will be used in the synthesized configuration script.

6.2.4 Resolving Configuration Task Ordering

Once we have a set of configuration tasks, we turn our attention to ordering them to produce an executable configuration script. Because configuration task dependencies are non-trivial to infer, we introduce an ordering routine based on error resolution. Synth starts its ordering procedure by converting the covering set of configuration tasks into a random sequence. It then runs the tasks in sequence order. When a configuration task fails, it queries its knowledge base for a matching error and the associated resolving tasks. The failed task is moved to the end of the sequence and the resolving tasks are inserted at its original location. Ordering then resumes with the first newly inserted task and continues until the entire sequence has been ordered. Tasks with failures that cannot be resolved by Synth are dropped from the ordering.

We refer to the configuration tasks from Figure 6.1 as an example. If Synth is reproducing a configuration script that installs and starts MySQL, it might generate an initial random ordering of the following configuration tasks:

```
1 systemctl start mysql
2 apt-get install -y mysql
3 apt-get update
```

During ordering, Synth will attempt to run the first `systemctl start` task and encounter the error:

```
ShellTaskError:
  exit_code: 5
  stdout: 'Failed to start mysql.service: Unit mysql.service not found.'
```

Synth will map the error in its knowledge base from Figure 6.1b to the error it has just encountered with the mapping from Figure 6.1d. While building the knowledge base, Synth recorded that the error could be resolved by running `apt-get update` and `apt-get install -y nginx`. It takes these resolving tasks and transforms them by applying the mapping to generate resolving tasks for MySQL. Finally, the tasks in the configuration script are reordered so that the resolving tasks appear in-order before the failed task, resulting in the final configuration script:

```
apt-get update
apt-get install -y mysql
systemctl start mysql
```

6.3 Evaluation

To evaluate Synth, we collected a golden dataset of 398 open-source Dockerfiles based on the official Debian and Ubuntu Docker images. We used the golden dataset as the ground-truth in our evaluation since it provides real-world configuration scenarios for the programs found in the Docker `ENTRYPOINT` and `CMD` statements. We assume that the images in the golden dataset are the "correct" environment configurations and that the behavior of the default commands within these images is the correct and expected behavior. This approach is related to the oracle problem from software testing and program repair, where generated solutions are measured against some pre-existing set of requirements that are assumed to define the desired behavior [Roj15; Lon16; Ton18; Ke15]. To avoid challenges with incomplete or incorrect Dockerfiles, we only include images that can build and successfully run the default command.

6.3.1 Docker Dataset

Our primary dataset is a collection of Docker images sourced from GitHub using Google BigQuery. We restricted our dataset to Dockerfiles that were based on the official Debian and Ubuntu images, used the default syntax, had defaults for all `ENV` and `ARG` values (and thus did not require user input), and did not rely on mounting secrets via the `RUN --mount` statement. We then cloned and built each Dockerfile, filtering those where the clone failed (often because they had been moved, deleted, or made private on GitHub), where the Dockerfile could not be parsed (because it used invalid syntax), where the default command succeeded in the base image (because the Dockerfile configuration would not affect the execution), where the image would not build (because we need a fully configured image), or where the default command did not succeed in the configured image (because the configuration was invalid).

```
FROM debian:buster
MAINTAINER Eric Ripa <eric@ripa.io>

RUN apt-get -q update \
    && apt-get -qy dist-upgrade \
    && apt-get -qy install opensmtpd \
    && rm -rf /var/lib/apt/lists/*

RUN echo "docker-smtpd" > /etc/mailname
RUN mkdir /etc/smtpd
COPY ./smtpd-template /etc/smtpd-template
COPY ./entrypoint.sh /

VOLUME ["/etc/smtpd", "/var/vmail"]
EXPOSE 25 587

ENTRYPOINT ["/entrypoint.sh"]
```

Figure 6.2 Example of one of the Dockerfiles in our experiment dataset from eripa/docker-opensmtpd-relay-debian. The final Docker image includes APT metadata and packages, configuration files, and directories.

Finally, we took a random sample of the Dockerfiles that remained and split them into test and training sets. The random sample was necessary to select a dataset size that we could feasibly build and evaluate. Prior studies, such as Shipwright, have noted the complexity and time requirements of building Dockerfiles on a large scale [Hen21]. This left us with 398 in the test set for our evaluation.

Figure 6.2 shows a valid Dockerfile from our dataset. The final Docker image that Synth will reproduce has several requirements for a valid configuration script:

1. The APT cache must be updated before installing packages.
2. The `opensmtpd` package must be installed.
3. The config file `/etc/mailname` must be written.
4. The directory `/etc/smtpd` must be created.

6.3.2 Synthesis Knowledge Base

We built the Synth knowledge base for our evaluation using the training dataset and a human-in-the-loop training process inspired by Shipwright [Hen21]. In this training process, Synth parsed all configuration tasks from the training dataset, clustered them by executable name, and presented the list of clusters to the practitioner participating in the training step. The practitioner then picked one or more representative examples from the cluster, or wrote new configuration tasks to act as representative examples, and provided them to Synth in the form of an analysis script.

The purpose of the human-in-the-loop approach is to allow a practitioner to aid Synth in reducing the size of its knowledge base during training by removing effective duplicates. While prior research has attempted to automatically determine when two configuration task invocations can perform the same action, further work is needed before we can do so reliably enough for training [Hor22a]. The shell command `apt-get` is a good example where further clustering and deduplication by a human is useful. APT is capable of updating caches, installing packages, removing packages, and more based on various subcommands and flags. It is also one of the most common configuration tasks in the training set. However, Synth only needs one configuration task template of each subcommand to effectively replicate its changes. Selecting just one example of `apt-get update`, `apt-get install`, etc. reduces the size of the Synth knowledge base and improves runtime performance.

Our training process resulted in 24 configuration task templates for common commands that Synth analyzed and recorded in its knowledge base. 15 of these had error resolution information. The most common error type was caused by trying to execute a command that was not installed on the system, such as:

```
1 exit_code: 127
2 stderr: npm: command not found
```

In each case, Synth recorded the sequence of commands from earlier in the configuration script that resulted in the command being installed.

6.3.3 Methodology

We evaluated Synth's ability to synthesize configuration scripts for the environments in our dataset from Section 6.3.1 using the knowledge base generated in Section 6.3.2 as follows. For each environment in the dataset:

1. Build the configured (ground truth) Docker image from the base Dockerfile.
2. Diff the configured and base images. This provides the desired change set provided to Synth for synthesis.
3. Synthesize a configuration script for the target configuration language using the desired change set.
4. Build the synthesized Docker image using the synthesized configuration script.
5. Diff the synthesized and base images. Use the diff to compute the Jaccard Coefficient.
6. Run the default command in the synthesized image and record the result.

Our synthesis procedure was run with a 30 minute timeout. At the timeout, synthesis was halted and marked as a failure because Synth had not yet produced an executable configuration script.

When executed the default command with a one minute timeout. Commands that timed out were considered a successful execution. This is due to the fact that many default commands in Docker images are designed to run indefinitely. For example, an image may be configured to run an ssh server that waits indefinitely for user connections after starting. Otherwise, the execution status of the default command was determined by the command's exit code. Any non-zero exit code was considered a failure. Any command that exited with status zero was considered a success.

Jaccard Coefficients were computed using the standard approach of taking the intersection over the union. In our case, the two change sets were the ones generated from the configured image and synthesized image diffs from the base image. We used exact equality for the intersection and union operations, not the change intersection algorithm from Section 6.2.3. This means that the Jaccard Coefficient represents how many changes exactly appeared in both the configured and synthesized images. A Jaccard Coefficient of 1.0 indicates that environment configurations were exactly the same, while a lower number indicates that the synthesized environment deviates in some way from the ground truth.

6.3.4 Experiment Hardware

Experiments were run on a cluster of 20 Digital Ocean servers using the storage optimized `so1_5-4vcpu-32gb` size slug which provides 4 CPUs, 32GB RAM, and a 900GB NVMe SSD. We ran local APT and PyPI proxies on each machine to improve performance.

6.4 Results

Synth finished synthesis within the timeout for 100 Dockerfiles. Overall, Synth enabled executability for many of the default commands in the evaluation when synthesis completed.

6.4.1 Dockerfiles

Synthesis timed out on 293 environments, leaving 100 fully synthesized configuration scripts. Timeouts were primarily due to the size of the environments. When synthesis completed, the average time required was 10.21 minutes. Overall, Synth's Dockerfiles enabled executability for 45/100 (45 %) of the default commands that failed in the base image. The synthesized images were highly similar to the ground truth configured images, with an average Jaccard Coefficient of 0.9558.

6.4.2 Synthesis Successes

6.4.2.1 APT Packages and Custom Configuration

One of the most common types of configuration from our dataset was the process of updating the APT cache, installing a package, removing the APT cache files, and then writing additional

```
FROM ubuntu:trusty
MAINTAINER Roman Zubatyuk <zubatyuk@gmail.com>

RUN apt-get update \
    && DEBIAN_FRONTEND=noninteractive apt-get install -y --no-install-recommends \
      dnsmasq dnsutils

RUN    echo "addn-hosts=/etc/althosts" > /etc/dnsmasq.conf \
    && echo "user=root" >> /etc/dnsmasq.conf \
    && echo "conf-dir=/etc/dnsmasq.d" >> /etc/dnsmasq.conf \
    && rm -rf /var/lib/apt/lists/* /var/cache/apt/*

CMD /usr/sbin/dnsmasq -d
```

Figure 6.3 Test Dockerfile from zubatyuk/docker-containers.

custom configuration. Figure 6.3 shows one of the Dockerfiles from our test set which does just that. The Dockerfile updates the system's APT cache, installs the `dnsmasq` and `dnsutils` packages, writes settings to the dnsmasq configuration file, and then removes the APT cache.

Synth generates a Dockerfile that performs all of these tasks (it also explicitly installs a number of transitive dependencies). An excerpt is shown below:

```
RUN apt-get update
RUN apt-get install -y --no-install-recommends dnsmasq
RUN apt-get install -y --no-install-recommends dnsutils
RUN rm -f /var/lib/apt/lists/archive.ubuntu.com_ubuntu_dists_trusty_Release.gpg
RUN rm -f /var/lib/apt/lists/archive.ubuntu.com_ubuntu_dists_trusty_Release
RUN rm -rf /var/cache/apt/archives
RUN rm -rf /var/cache/apt/archives/partial
RUN rm -f /var/lib/apt/lists/lock
RUN rm -f /var/cache/apt/archives/lock
RUN echo -n $'addn-hosts=/etc/althosts\nuser=root\nconf-dir=/etc/dnsmasq.d\n' >>
↪ /etc/dnsmasq.conf
RUN rm -rf /var/lib/apt/lists/partial
```

Notably, neither of the installed packages appear in the Synth knowledge base, nor do the dnsmasq configuration file or values. Synth is able to build a working configuration script by mapping the configuration task templates it has recorded to the set of desired changes. The template and mapping values are used to produce the final configuration task definitions. The ordering of `apt-get update` before `apt-get install` is enforced by Synth's ordering and error resolution routine.

6.4.2.2 Custom Users and Groups

Some Docker images in our evaluation contained custom user or group definitions. This can happen as the result of installing a package (such as in slok/favorshare-dockerfiles[1]) or explicitly as part of the configuration script (such as sukramlitt/docker-shellinabox[2]).

Adding a group or user account is a complicated process that involves writing to several Linux `/etc` configuration files. In addition, user account creation typically involves creating the user's home directory structure with default shell configuration files. In both cases, Synth is able to correctly inspect the system configuration and generates the configuration tasks

```
1 RUN addgroup redis
```

and

```
1 RUN adduser shellboxuser
```

as part of the respective configuration scripts.

6.4.2.3 Partial Successes

Synth was able to discover and replicate important configurations even when the synthesized Dockerfile did not enable executability of the default command. Table 6.1 highlights interesting cases where this occurs. These Dockerfiles can provide a starting place for developers.

6.4.3 Synthesis Failures

We performed a manual review of the 55 synthesized Dockerfiles that did not enable executability for the default command to determine why they failed. Failures belonged to five categories. Additional failure examples are shown in Table 6.2.

6.4.3.1 Missing Download

26 images contained an executable that was downloaded from an external source. Because the original download URL is not preserved as part of the system changes, Synth is unable to determine where the executable should be sourced from and cannot match it to a template for a configuration task like `curl`. While Synth does support writing arbitrary text files based on the configuration task templates it has, that support has not been extended to arbitrary binary executable files. Doing so would allow Synth to bypass the download step and correctly write the binary file at the final location.

[1] `https://github.com/slok/favorshare-dockerfiles/blob/469001341f25edce98f38b3e9d197bcd63da6fec/redis/Dockerfile`

[2] `https://github.com/sukramlitt/docker-shellinabox/blob/9f468e6e2bb9d0aa2058bdf6d73d784d6e4c5ac8/Dockerfile`

Table 6.1 Examples of successful configuration tasks generated by Synth.

Dockerfile	Configuration Task	Explanation
imos/container	`adduser cloud-guest`	The original Dockerfile runs `useradd` with the `--create-home` flag. Synth has picked a different utility that also creates the user with a home directory.
marcermarc/DockerSteam	`addgroup steam`	Synth detected changes related to the group creation and picked the same configuration task for its synthesized script as in the original Dockerfile.
masatoshiitoh/dockerfiles	`ln -sf /etc/ssl/certs/ ⌋ ↪ ssl-cert-snakeoil ⌋ ↪ .pem ↪ /etc/ssl/certs/ ⌋ ↪ eacf3c0a`	masatoshiitoh/dockerfiles installs `yaws` which has a transitive dependency on `ssl-cert`. During synthesis, Synth detects the changes made by the `ssl-cert` installation and manually creates a symbolic link for the cert to the correct destination.
romeOz/docker-nginx	`echo '...' >> ↪ /etc/locale.gen && ↪ locale-gen`	The source Dockerfile generates locales for `en_US.UTF-8`. Synth correctly writes the config value to `/etc/locale.gen` and runs `locale-gen`.
iwanders/vpn_infrastructure	`mkdir -p ↪ /etc/nginx/certs`	Docker creates the `/etc/nginx/certs` directory indirectly via `ADD ./etc/ /etc/`. The statement copies the directory structure from the local project into the final image. Synth is able to replicate the directory creation using a different configuration task.
phpdocker-io/base-images	`rm -f /usr/share/doc/ ⌋ ↪ base-files/copyright`	Synth correctly removes this and other documentation files that come from running `apt-get install`. The original Dockerfile does so with `rm -rf /usr/share/doc/*`.

6.4.3.2 No Configuration Task Selected

17 images were missing a necessary configuration because Synth failed to include a configuration task that covered the changes in the final script. This could happen because of a bad match for the existing templates in the knowledge base, because Synth had not been trained on the configuration task, or because of an error that Synth did not know how to resolve. These failures could be resolved with additional examples and training or improvements to the mapping process.

6.4.3.3 Incorrect Configuration Task Selected

In 4 cases, Synth picked a configuration task that made too many changes (the intended configuration changes, plus some extras that wouldn't have been made by the correct command). The extra changes resulted in the default command failing. This could occur, for example, when extra files were written to a configuration directory, causing the default command to fail while parsing them.

Table 6.2 Synthesis failures corresponding to main failure categories.

Dockerfile	Executable	Error Message	Failure Reason
pact-foundation/pact-stub-server	`./pact-stub-server`	./pact-stub-server: command not found	Synth did not know how to download the pact-stub-server binary.
envoyproxy/envoy	`python3`	...SyntaxError: invalid syntax	Synth failed to use the correct `pip install` command. It later tried to manually write the Python dependency files using `echo`, but the task resulted in incorrect syntax.
manicmonkey/licensing	`nginx`	nginx: [emerg] unexpected end of file	Synth erroneously used `git clone <url> <dir>` to create a directory instead of `mkdir <dir>`. This created incorrectly formatted files within the configuration directory.
egorpe/cloud-cd-eval	`nginx`	nginx: [emerg] "user" directive is duplicate in /etc/nginx/nginx.conf:96	Synth wrote to the nginx config file using `echo`, but duplicated a section from the default file contents.

6.4.3.4 Incorrect Changes from a Synthesized Configuration Task

In 3 other cases, Synth picked a configuration task to make a desired change, but the final task definition had an incorrect effect on the system. The instance from Table 6.2 happened when Synth wrote a change to the nginx config file, but the change contents conflicted with the default file contents that came from installing nginx.

6.4.3.5 Flakiness

5 images failed due to flakiness and runtime errors.

6.5 Limitations

Ultimately, Synth's ability to infer which configuration tasks should be used in a configuration script is dependent on its ability to know about and match task behaviors to desired changes. While this is incredibly useful, any missing information or bad matches will result in missing or invalid configurations.

6.5.1 Knowledge Gathering

Synth's configuration scripts are limited by the configuration tasks in its knowledge base. Because of this, it is tempting to use its ability to observe and record information about configuration

tasks to build a large knowledge base from many existing configuration scripts. Synthesis can certainly benefit from having a larger number of available configuration tasks and a larger number of examples of each to work from, since having more examples yields a higher likelihood of having an exact match or being able to map between a recorded and desired change correctly.

However, a larger knowledge base has a measurable impact on execution time. The Configuration Cover subroutine searches for a covering from all available configuration task templates. The larger the knowledge base is, the longer it will take for synthesis to finish because Synth must consider every known configuration task at least once. More examples also means a higher likelihood of bad matches that lead to incorrectly configured environments. These drawbacks make the act of knowledge gathering an important but challenging step in the synthesis pipeline.

6.5.2 Dealing with Close Matches

Because Synth cannot feasibly have an exact match for every desired configuration change in its knowledge base, its synthesis routine is heavily reliant on the mapping algorithm described in Section 6.2.3. The mapping algorithm is shown to work extremely well for configuration tasks in our evaluation and is the reason why Synth is able to install arbitrary packages from different package managers, link file paths, and write text with only a single example of the relevant task. But, mapping can be broken when the value of the desired configuration change does not quite align with the changes Synth knows how to make.

The most common case of broken mappings occurs when using Synth to translate a configuration to a different configuration system. While systems like Docker and Ansible have configuration changes that make analogous changes, they may do so in slightly different ways. For example, using a shell script, one might append multiple lines to a file using the command:

```
cat <<EOF >> file.txt
    line 1
    line 2
EOF
```

In Ansible, one way to write multiple lines to a file is with the `blockinfile` module:

```
ansible.builtin.blockinfile:
  path: file.txt
  block: |
    line 1
    line 2
```

However, this results in writing the following lines:

```
# BEGIN ANSIBLE MANAGED BLOCK
line 1
line 2
# END ANSIBLE MANAGED BLOCK
```

The extra text surrounding the two lines prevents Synth from mapping the Ansible module's change to the desired change produced by the shell command.

6.5.3 Recognizing Argument Usage for Templates

Configuration task templates are a generic representation of the changes that a configuration task can make when provided with different user parameters. Templates often work quite well, as many configuration tasks are designed to accept user parameters that appear verbatim in the changes that they cause to the system under configuration. When arguments do not appear verbatim in changes, though, Synth can struggle to build a correct template. The following change definition from executing `git config --global user.name username` demonstrates this:

```
1  FileChange:
2    path = '/root/.gitconfig'
3    changes = {
4      FileContentChange:
5        change_type = 'addition'
6        content = '[user]\n\tname = username\n'
7    }
```

Although the git configuration key and value appear in the configuration file, the key does not appear verbatim. Instead it is split into the `[user]` section and the subsequent `name` attribute. This prevents Synth from extracting it during template construction. Later, it will fail to correctly map the `user.name` value to other git configuration keys. Future work should focus on methods for improving template construction and inferring when a template value includes a user argument.

6.5.4 External Sources

Our synthesis procedure expects that configuration tasks have a predictable and repeatable effect on the system. This is very often true, since random configurations are typically not helpful for developers. However, if Synth has trouble determining what changes a task will have, then it may not be able to correctly select and format that task for use in a configuration script.

The shell commands `curl` and `wget` are good examples where an effect is hard to predict. The change to the system is dependent on the URL and the server being contacted, but the URL itself is not recorded as a part of the change. Synth is therefore unable to map a file change back to source URL unless that exact URL and file change have been seen before.

6.5.5 Intermediate Configuration States

One of the biggest challenges for Synth is dealing with intermediate states when one or more commands must be run in sequence to produce a desired outcome. Synth is able to use its task ordering and error resolution procedure to infer sequences in cases where it has seen the error before. However, some configuration tasks don't produce an error when run out of order. For

example, the `pip` package manager supports setting the `index-url` value in its config file to change where packages are installed from. The install may still work if `pip install` is run before the config file is updated, but the package will be installed from the wrong package index. Writing the index url is a necessary intermediate state that cannot be inferred by error resolution.

6.6 Future Work

Our results show that Synth is a promising approach to synthesizing configuration scripts that produce desired environment configurations. While just being able to synthesize a configuration script for an environment is a useful result, additional work can improve the quality of synthesized configuration scripts through minimization, increasing their readability, and improving their performance.

6.6.1 Configuration Script Minimization

Automatically generated programs can be difficult to understand due to their size and complexity. The software testing community addresses this problem with test case minimization, which reduces a test case to the smallest possible size while preserving the desired functionality [Lei07]. More recently, similar approaches have been applied to computing environments to reduce their size, complexity, and attack surface for security vulnerabilities [Ras17a; Ras17b; Qia19; Koo19].

Because Synth generates the configuration script for an environment, we believe that it would be possible to both minimize the configuration script and the final image by correctly pruning or altering configuration tasks before the final script is returned. Pruning can likely be accomplished by a delta-debugging algorithm to remove tasks that have no effect, are subsumed by another (such as installing a package and its transitive dependencies), or whose changes are not needed for successful program execution [Zel02]. In addition, some configuration tasks like `pip install` can have multiple invocations combined into a single command that may be more concise and easier to read.

6.6.2 Readability and Maintainability

Previous work has demonstrated the need for automated code generation tools to focus on usability and developer understanding [Roj15]. The same is true for configuration scripts, as developers must be able to read and maintain their configurations. Synth does not make readability and maintainability a primary focus, but we believe that future efforts should focus on extending work on normalization and generalization, descriptive names, good comments, and accompanying synthesized documentation to configuration scripts [Gro17; Dak17; Fry12].

6.6.3 Configuration as a Search Problem

Synth fundamentally treats the problem of configuration as an instance of Set Cover with an extension that results in an ordered sequence. While this is broadly applicable to many configuration tasks, it can miss intermediate states during configuration that are required. To overcome this challenge and achieve more accurate and more complicated configuration scripts, we believe that future work may need to 1) model the system configuration that a configuration task relies on in addition to the configuration changes that it produces, 2) introduce theories for handling data passed to stdin, stdout, stderr in addition to task arguments, 3) handle variables, and 4) model the propagation of information through configuration tasks to allow for chained/piped tasks.

If future work can solve these challenges, it should be feasible to treat configuration synthesis as a standard search problem where the starting state is the base environment and the target state is the desired environment. In this paradigm, a state transition would be the result of picking a configuration task that is compatible with the current system state, the outputs of the previous command, or available variables to produce some new desired change. The final configuration script would be the sequence of state transitions that leads from the base to the desired environment. Optimal configurations may lay along the shortest paths to the target.

6.7 Discussion

Developers engage in configuration management practices because programs rely on certain aspects of the system state to run correctly. While these practices are necessary, they are also challenging, time consuming, and potentially costly to get wrong [Idc15; Git; Sev14]. Configuration synthesis as a domain promises developers assistance with automatically creating and maintaining configuration scripts and/or entire system configurations.

There are several broad applications of synthesis. The first, which Synth begins to address, is the problem of reconstructing a configuration script from some representation of the final environment (a set of changes, in our case). Prior research has only focused on synthesizing patches for existing configuration scripts [Wei17]. Unlike prior approaches, full synthesis can also be applied to legacy environments for which no configuration script exists and where nobody remembers how the system was configured in the first place. On a slightly smaller scale, it can address the "works on my machine" problem by helping developers synthesize and distribute their working environment configurations for development.

Configuration synthesis can also aid developers in translating their configuration scripts from one configuration management platform to another. Prior approaches have attempted to translate configuration management tasks individually, but have faced challenges translating entire configuration scripts [Hor22a]. We believe that translation should be feasible by first running a configuration script to produce an environment configuration and then feeding that configuration to a tool like Synth for output in a different configuration language. In this way,

the environment configuration itself becomes an intermediary representation that is used to carry information between the configuration management systems. Section 6.7.1 discusses an exploratory evaluation we performed using Synth to translate Dockerfiles to Ansible playbooks.

Finally, configuration synthesis has the potential to aid developers by directly satisfying program needs. We expand upon our thoughts and the potential applications in Section 6.7.2.

6.7.1 Translating to Ansible Playbooks

We performed an exploratory evaluation of translating the environment configurations in our experiment dataset to Ansible playbooks by way of synthesis. The evaluation used the same procedure as our Dockerfile evaluation, but we trained Synth by manually translating the representative tasks for Docker into their Ansible equivalents (Synth could also run a human-in-the-loop training step with a dedicated Ansible training dataset) and used a 2.5 hour timeout. The additional time was provided because Ansible environments must contain Ansible and all of its dependencies, a requirement that impacts runtime performance.

The evaluation resulted in 25 fully synthesized playbooks. There were two primary causes for Synth being unable to synthesize a playbook. The first was a challenge with compatibility between the OS image, Python support, and Ansible support. Environments based on Ubuntu 14 (trusty) were shown to fail when Synth tried to run Ansible modules for ordering with the error message:

```
Ansible requires a minimum of Python2 version 2.6 or Python3 version 3.5. Current version:
↪  3.4.3 (default, Nov 12 2018, 22:25:49) [GCC 4.8.4]
```

Other failures were due to a standard timeout at the time limit. When synthesis did complete for playbooks, the average time required was 28.65 minutes.

Synth's Ansible playbooks enabled executability for 14/25 (56 %) of the default commands. The synthesized images were highly similar to the ground truth configured images, with an average Jaccard Coefficient of 0.9652.

Figure 6.4 shows an ubuntu:focal Dockerfile in our dataset from vicamo/docker-binfmt-qemu.[3] The image's default command executes `update-binfmts` and `dpkg-reconfigure`. This command will fail without the binfmt-support and qemu-user-static packages which are installed by the configuration script. Figure 6.5 shows the synthesized Ansible playbook from our evaluation. The playbook correctly uses Ansible's `ansible.builtin.apt` module to install binfmt-support and qemu-user-static from the APT ecosystem. It also explicitly installs two transitive dependencies whose changes were present and covered as part of the desired change set. Notably, the original change set provided to Synth was for the Docker image produced by a Dockerfile. The synthesis process has effectively translated the Dockerfile into an Ansible playbook by using the system configuration as an intermediate representation. Additional work should focus on the challenges of translating representations between configuration languages.

[3] `https://github.com/vicamo/docker-binfmt-qemu`

```
FROM ubuntu:focal

MAINTAINER vicamo@gmail.com

RUN apt-get update \
        && apt-get install -y --no-install-recommends \
                binfmt-support \
                qemu-user-static \
        && apt-get clean \
        && rm -rf /var/lib/apt/lists/*_dists_*

ENTRYPOINT ["sh", "-c", "update-binfmts --enable && dpkg-reconfigure qemu-user-static"]
```

Figure 6.4 Source Dockerfile from vicamo/docker-binfmt-qemu. The Dockerfile configures an image that installs binfmt-support and qemu-user-static and sets the `ENTRYPOINT`. The default command fails in an unconfigured debian:bullseye image.

```
- become: true
  hosts: localhost
  tasks:
  - ansible.builtin.apt:
      install_recommends: false
      name: libpipeline1
      state: present
    name: Run ansible.builtin.apt
  - ansible.builtin.apt:
      install_recommends: false
      name: binfmt-support
      state: present
    name: Run ansible.builtin.apt
  - ansible.builtin.apt:
      update_cache: true
    name: Run ansible.builtin.apt
  - ansible.builtin.apt:
      install_recommends: false
      name: libpipeline1:amd64
      state: present
    name: Run ansible.builtin.apt
  - ansible.builtin.apt:
      install_recommends: false
      name: qemu-user-static
      state: present
    name: Run ansible.builtin.apt
  - ...
```

Figure 6.5 Synth's playbook that reproduces the environment from Figure 6.4. In addition to installing the two required packages, Synth installs some transitive dependencies explicitly as a result of its search process.

6.7.2 Synthesizing Configuration Scripts Based on Program Needs

Our approach to Synth is able to answer the basic question, "Given a set of desired configuration changes, how do I reproduce them?" While this is useful in its own right, developers do not always have an example of a working environment or know what changes need to be made. We believe that there is a natural extension to the synthesis problem that can be phrased as, "Given a non-working program, what configuration changes do I need to make for it to become executable?"

If future work can solve the latter problem, then it should naturally combine with configuration synthesis techniques like Synth's to enable configuration script synthesis based on program needs. Such a tool would first inspect or execute a program to find instances where it interacts with the system. Any failures or attempted interactions with a resource that does not exist would indicate places where a configuration change would need to be made. In aggregate, these changes could be used as the final system specification used for configuration synthesis. This would present an improvement over prior work like DockerizeMe and V2, where the technique is dedicated to only a single type of configuration [Hor19a; Hor19b].

6.8 Related Work

Much of the recent work on configurations is focused on optimizing "internal" application configuration for performance, security, stability, or other application metrics rather than focusing on the "external" configuration of system state [Ngu19; Che20b; Ngu21; Nai18]. When research does focus on system state, it usually aims to assist developers with configuration management tasks instead of fully generating configurations. For example, Raab et al. recently introduced Elektra, a configuration management tool for universally managing configuration file values via Puppet [Raa20]. They show that Elektra increases developer productivity.

In the system configuration space, Stöckle et al. highlight that security misconfigurations are primarily due to lack of knowledge [Stö20]. They address this issue in the same general way that we do by proposing a scheme to automatically synthesize Windows security settings. Their approach differs from ours in that they use natural language processing to infer available settings from configuration guides and validate their settings against Windows administrative template files. By contrast, our configuration tasks are inferred from observing the system state and validated by executing against the system.

Nelson et al. also tackle configuration synthesis by using state-machine models [Nel19]. While this approach may prove to be more sound, it faces significant drawbacks in terms of performance and effort. Formally solving for system state is a computationally intensive task and building a proper model of system state requires significant effort from administrators.

Other research also takes advantage of error information in ways similar to Synth. Ni et al. introduce SOAR, a tool for refactoring API usage via program synthesis that uses error messages to generate additional constraints on inputs and thus narrow their search space [Ni21]. D'Antoni

et al. introduce NoFAQ, a tool for repairing uses of command line tools. Their handling of errors is perhaps the most similar to ours. NoFAQ is able to find fixes by matching an error message to a previously observed error message and its associated fix [D'A17]. Our approach improves on their technique by generalizing the matching process between objects, allowing us to match full sets of configuration changes in addition to errors.

6.9 Conclusion

We present Synth, a technique for synthesizing configuration scripts that produce a specified environment. Synth operates by treating the configuration synthesis problem as an instance of set cover and finding the set of configuration tasks that fully covers the set of desired changes. The correct task order is inferred by executing the configuration tasks and resolving errors. The data for synthesis comes from observing the execution of configuration tasks on real systems. Our evaluation shows that Synth is capable of reproducing environment configurations that enable program executability.

CHAPTER

7

CONCLUSION

This work investigates automated techniques for inferring and generating computing environments through the use of configuration scripts. The primary purpose is to assist developers with performing the configuration management tasks that are necessary to enable program execution and support the deployment and maintenance of code. Our techniques are fundamentally based on the idea that we can gather knowledge about configuration management either by inspecting pre-existing configured environments and their corresponding configuration scripts or by systematically executing and observing different units of configuration to build a picture of a larger configuration management system. With a sufficient amount of knowledge gathering, the correct representation in a knowledge base, and a targeted search/inference algorithm, we show that satisfying program needs via automated configuration management becomes feasible.

There are, of course, still challenges in the automated configuration domain that must be addressed. Current techniques are still limited by their ability to gather, represent, and store information about possible configurations in ways that enable efficient inference algorithms. Many are also based on a generate-and-validate approach which is inherently costly, especially at the scale of building and validating full system images. In addition to solving for program executability, automated configuration management should solve for availability between interconnecting services, security, detection and recovery from configuration errors, and more. Future work will undoubtedly build on the work presented in this thesis, and by others, to address and overcome these challenges.

BIBLIOGRAPHY

[Haca] https://news.ycombinator.com/item?id=5933400. 2013.

[Hacb] https://news.ycombinator.com/item?id=20379217. 2019.

[Hacc] https://news.ycombinator.com/item?id=20375380. 2020.

[Ans] *Ansible*. https://www.ansible.com/. Red Hat.

[Ans16] Ansible. *Nasa: Increasing Cloud Efficiency With Ansible And Ansible Tower*. https://www.ansible.com/hs-fs/hub/330046/file-1649288715-pdf/Whitepapers__Case_Studies/nasa_ansible_case_study.pdf. 2016.

[Staa] *Ansible and Playbook. How to convert shell commands into yaml syntax?* https://stackoverflow.com/questions/21005290/ansible-and-playbook-how-to-convert-shell-commands-into-yaml-syntax. 2014.

[Reda] *Ansible versus BASH script*. https://www.reddit.com/r/linuxadmin/comments/emcuqm/ansible_versus_bash_script/. 2020.

[Bar15] Barr, E. T. et al. "The Oracle Problem in Software Testing: A Survey". *IEEE Transactions on Software Engineering* **41**.5 (2015), pp. 507–525.

[Redb] *Bash scripts to Ansible*. https://www.reddit.com/r/ansible/comments/a1qpr0/bash_scripts_to_ansible/. 2018.

[Bec18] Becker, B. A. et al. "Fix the First, Ignore the Rest: Dealing with Multiple Compiler Error Messages". *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. Sigcse '18. New York, NY, USA: Acm, 2018, pp. 634–639.

[Bir08] Birks, M. et al. "Memoing in qualitative research: Probing data and processes". *Journal of Research in Nursing* **13**.1 (2008), pp. 68–75.

[Bog15] Bogart, C. et al. "When It Breaks, It Breaks: How Ecosystem Developers Reason about the Stability of Dependencies". *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 2015, pp. 86–89.

[Cam13] Campbell, J. L. et al. "Coding in-depth semistructured interviews". *Sociological Methods & Research* **42**.3 (2013), pp. 294–320.

[Che] *Chef*. https://www.chef.io/. Progress.

[Che20a] Chen, C. "SimilarAPI: Mining Analogical APIs for Library Migration". *Proceedings of the 42nd International Conference on Software Engineering*. Icse '20. 2020.

[Che20b] Chen, Q. et al. "Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems". *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020, 362–374.

[Cit17] Cito, J. et al. "An Empirical Analysis of the Docker Container Ecosystem on GitHub". *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, pp. 323–333.

[Col] Collins, R. *PEP 508 – Dependency specification for Python Software Packages*. `https://www.python.org/dev/peps/pep-0508/`. Online; Accessed August 12, 2018.

[Fre] *Convert shell scripts to ansible playbook*. `https://www.freelancer.com/projects/python/convert-shell-scripts-ansible-playbook/`.

[Stab] *Converting shell script to Ansible play*. `https://stackoverflow.com/questions/47460608/converting-shell-script-to-ansible-play`. 2017.

[Dag11] Dagenais, B. & Robillard, M. P. "Recommending Adaptive Changes for Framework Evolution". *ACM Trans. Softw. Eng. Methodol.* **20**.4 (2011), 19:1–19:35.

[Dak17] Daka, E. et al. "Generating Unit Tests with Descriptive Names or: Would You Name Your Children Thing1 and Thing2?" *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Issta 2017. New York, NY, USA: Association for Computing Machinery, 2017, 57–67.

[D'A17] D'Antoni, L. et al. "NoFAQ: Synthesizing Command Repairs from Examples". *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Esec/fse 2017. New York, NY, USA: Association for Computing Machinery, 2017, 582–592.

[Dec17] Decan, A. et al. "An empirical comparison of dependency issues in OSS packaging ecosystems". *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017, pp. 2–12.

[DeK14] DeKoenigsburg, G. *Case Study: Ansible and NASA*. `https://www.slideshare.net/AllThingsOpen/case-study-ansible-and-nasa`. 2014.

[Die19] Dietrich, J. et al. "Dependency Versioning in the Wild". 2019.

[Eas18] Eastwood, A. *Shell Scripts to Ansible*. `https://www.ansible.com/blog/shell-scripts-to-ansible`. 2018.

[Edd13] Eddy, B. P. et al. "Evaluating source code summarization techniques: Replication and expansion". *2013 21st International Conference on Program Comprehension (ICPC)*. 2013, pp. 13–22.

[Stac] *Execute curl command in ansible*. `https://stackoverflow.com/questions/59983700/execute-curl-command-in-ansible`. 2020.

[Fan16] Fang-Hsiang Su et al. "Identifying functionally similar code in complex codebases". *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 2016, pp. 1–10.

[Sen] *Flask*. `https://docs.sentry.io/clients/python/integrations/flask/`. 2018.

[For16] Ford, D. et al. “Paradise Unplugged: Identifying Barriers for Female Participation on Stack Overflow”. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Fse 2016. New York, NY, USA: Acm, 2016, pp. 846–857.

[Fry12] Fry, Z. P. et al. “A Human Study of Patch Maintainability”. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. Issta 2012. New York, NY, USA: Association for Computing Machinery, 2012, 177–187.

[Ger07] German, D. M. et al. “A Model to Understand the Building and Running Inter-Dependencies of Software”. *14th Working Conference on Reverse Engineering (WCRE 2007)*. 2007, pp. 140–149.

[Git] GitLab. *Postmortem of Database Outage of January 31*. `https://about.gitlab.com/2017/02/10/postmortem-of-database-outage-of-january-31/`.

[Gro17] Groce, A. et al. “One Test to Rule Them All”. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Issta 2017. New York, NY, USA: Association for Computing Machinery, 2017, 1–11.

[Hai10] Haiduc, S. et al. “On the Use of Automated Text Summarization Techniques for Summarizing Source Code”. *Proceedings of the 2010 17th Working Conference on Reverse Engineering*. Wcre ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 35–44.

[Has18] Hassan, F. & Wang, X. “HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts”. *Icse*. Icse 2018. 2018.

[Hej18] Hejderup, J. et al. “Software Ecosystem Call Graph for Dependency Management”. *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. Icse-nier ’18. New York, NY, USA: Acm, 2018, pp. 101–104.

[Hen05] Henkel, J. & Diwan, A. “CatchUp!: Capturing and Replaying Refactorings to Support API Evolution”. *Proceedings of the 27th International Conference on Software Engineering*. Icse ’05. New York, NY, USA: Acm, 2005, pp. 274–283.

[Hen21] Henkel, J. et al. “Shipwright: A Human-in-the-Loop System for Dockerfile Repair”. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2021, pp. 198–199.

[Hoc15] Hochstein, L. `https://twitter.com/lhochstein/status/679731676193230849`. 2015.

[Hor17] Horowitz, J. *Configuration Management is an Antipattern*. `https://hackernoon.com/configuration-management-is-an-antipattern-e677e34be64c`. 2017.

[Hor18] Horton, E. & Parnin, C. “Gistable: Evaluating the Executability of Python Code Snippets on GitHub”. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 217–227.

[Hor19a] Horton, E. & Parnin, C. "DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets". *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 328–338.

[Hor19b] Horton, E. & Parnin, C. "V2: Fast Detection of Configuration Drift in Python". *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 477–488.

[Hor22a] Horton, E. & Parnin, C. "Dozer: Migrating Shell Commands to Ansible Modules via Execution Profiling and Synthesis". *44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '22)*. 2022.

[Hor22b] Horton, E. & Parnin, C. "Synthesizing Configuration Scripts". *2022 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2022.

[Redc] *How to convert this bash SED command to ansible?* `https://www.reddit.com/r/ansible/comments/95bb2z/how_to_convert_this_bash_sed_command_to_ansible/`. 2018.

[Fiv] *I will convert bash scripts to ansible*. `https://www.fiverr.com/rzaluska/convert-bash-scripts-to-ansible-c81d`.

[Idc15] Idc. *DevOps and the Cost of Downtime*. `https://developer.ibm.com/urbancode/docs/devops-and-the-cost-of-downtime-fortune-1000-best-practice-metrics-quantified/`. 2015.

[Jay13] Jaynes, M. *Shell Scripts vs Ansible: Fight!* `https://hvops.com/articles/ansible-vs-shell-scripts/`. 2013.

[Jia15] Jiang, Y. & Adams, B. "Co-evolution of Infrastructure and Source Code - An Empirical Study". *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 2015, pp. 45–55.

[Kal14] Kalliamvakou, E. et al. "The Promises and Perils of Mining GitHub". *Proceedings of the 11th Working Conference on Mining Software Repositories*. Msr 2014. New York, NY, USA: Acm, 2014, pp. 92–101.

[Kar72] Karp, R. M. "Reducibility among Combinatorial Problems". *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Miller, R. E. et al. Boston, MA: Springer US, 1972, pp. 85–103.

[Ke15] Ke, Y. et al. "Repairing Programs with Semantic Code Search (T)". *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2015, pp. 295–306.

[Koo19] Koo, H. et al. "Configuration-Driven Software Debloating". *Proceedings of the 12th European Workshop on Systems Security*. EuroSec '19. New York, NY, USA: Association for Computing Machinery, 2019.

[Le16] Le, X. B. D. et al. "History Driven Program Repair". *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 213–224.

[Lei07] Leitner, A. et al. "Efficient Unit Test Case Minimization". *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*. Ase '07. New York, NY, USA: Association for Computing Machinery, 2007, 417–420.

[Lia98] Liang, D. & Harrold, M. J. "Slicing objects using system dependence graphs". *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 1998, pp. 358–367.

[Lon16] Long, F. & Rinard, M. "Automatic Patch Generation by Learning Correct Code". *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Popl '16. New York, NY, USA: Association for Computing Machinery, 2016, 298–312.

[Luc14] Lucia, A. et al. "Labeling Source Code with Information Retrieval Methods: An Empirical Study". *Empirical Softw. Engg.* **19**.5 (2014), pp. 1383–1420.

[Lun10] Lungu, M. et al. "Recovering Inter-project Dependencies in Software Ecosystems". *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. Ase '10. New York, NY, USA: Acm, 2010, pp. 309–312.

[M.17] M., S. *Error installing pytorch vision*. `https://github.com/pytorch/pytorch/issues/2231`. 2017.

[Mac18] Macho, C. et al. "Automatically repairing dependency-related build breakage". *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 106–117.

[McB14] McBurney, P. W. et al. "Improving Topic Model Source Code Summarization". *Proceedings of the 22Nd International Conference on Program Comprehension*. Icpc 2014. New York, NY, USA: Acm, 2014, pp. 291–294.

[McI11] McIntosh, S. et al. "An Empirical Study of Build Maintenance Effort". *Proceedings of the 33rd International Conference on Software Engineering*. Icse '11. New York, NY, USA: Acm, 2011, pp. 141–150.

[Mir17] Mirhosseini, S. & Parnin, C. "Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-date Dependencies?" *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. Ase 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 84–94.

[Nai18] Nair, V. et al. "Finding Faster Configurations using FLASH". *IEEE Transactions on Software Engineering* **Pp** (2018).

[Redd] *Need help converting bash to ansible*. https://www.reddit.com/r/ansible/comments/9f615c/need_help_converting_bash_to_ansible/. 2018.

[Nel19] Nelson, T. et al. "Synthesizing Mutable Configurations: Setting up Systems for Success". *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 2019, pp. 81–85.

[Ngu21] Nguyen, K. & Nguyen, T. "GenTree: Inferring Configuration Interactions using Decision Trees". *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021, pp. 1232–1236.

[Ngu19] Nguyen, S. "Configuration-Dependent Fault Localization". *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. Icse '19. IEEE Press, 2019, 156–158.

[Ngu16] Nguyen, T. D. et al. "Mapping API Elements for Code Migration with Vector Representations". *Proceedings of the 38th International Conference on Software Engineering Companion*. Icse '16. New York, NY, USA: Association for Computing Machinery, 2016, 756–758.

[Ni21] Ni, A. et al. "SOAR: A Synthesis Approach for Data Science API Refactoring". *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021, pp. 112–124.

[Pupa] *NYSE and ICE: Compliance, DevOps and Efficient Growth with Puppet Enterprise*. https://media.webteam.puppet.com/uploads/2019/11/puppet-CS-NYSE-ICE.pdf. 2019.

[Par13] Parnin, C. et al. "Blogging developer knowledge: Motivations, challenges, and future directions". *2013 21st International Conference on Program Comprehension (ICPC)*. 2013, pp. 211–214.

[Par12] Parnin, C. et al. "Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow". *Georgia Institute of Technology, Tech. Rep* (2012).

[Pat13] Patterson, A. https://twitter.com/tenderlove/status/381893193949667328. 2013.

[Pet] Peterson, B. *PEP 373 – Python 2.7 Release Schedule*. https://www.python.org/dev/peps/pep-0373/. Online; Accessed May 7, 2019.

[Pim19] Pimentel, J. F. et al. "A Large-scale Study about Quality and Reproducibility of Jupyter Notebooks". *Proceedings of the 16th International Conference on Mining Software Repositories*. Msr '19. 2019.

[Pon06] Ponterotto, J. "Brief note on the origins, evolution, and meaning of the qualitative research concept thick description". *The Qualitative Report* **11**.3 (2006).

[Pupb] *Puppet*. https://puppet.com/. Puppet.

[Pyh] *PyHum*. https://github.com/dbuscombe-usgs/PyHum. 2018.

[Pyp] *Python Package Index*. https://pypi.org/. 2018.

[Qia19] Qian, C. et al. "RAZOR: A Framework for Post-deployment Software Debloating". *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019, pp. 1733–1750.

[Raa20] Raab, M. et al. "Unified Configuration Setting Access in Configuration Management Systems". *Proceedings of the 28th International Conference on Program Comprehension*. New York, NY, USA: Association for Computing Machinery, 2020, 331–341.

[Rah18] Rahman, A. & Williams, L. "Characterizing Defective Configuration Scripts Used for Continuous Deployment". *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 2018, pp. 34–45.

[Rah18] Rahman, A. et al. "What Questions Do Programmers Ask about Configuration as Code?" *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*. RCoSE '18. New York, NY, USA: Association for Computing Machinery, 2018, 16–22.

[Ras17a] Rastogi, V. et al. "Cimplifier: Automatically Debloating Containers". *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Esec/fse 2017. New York, NY, USA: Acm, 2017, pp. 476–486.

[Ras17b] Rastogi, V. et al. "New Directions for Container Debloating". *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*. Feast '17. New York, NY, USA: Association for Computing Machinery, 2017, 51–56.

[Rau18] Raugh, M. *Cutting The Strings: Migrating From Puppet Enterprise To Ansible Tower*. https://www.ansible.com/migrating-from-puppet-enterprise-to-ansible-tower. 2018.

[Raw18] Rawlins, L. *How to get started using Ansible*. https://sudoedit.com/how-to-get-started-using-ansible/. 2018.

[Ren18] Ren, Z. et al. "Automated Localization for Unreproducible Builds". *Proceedings of the 40th International Conference on Software Engineering*. Icse '18. New York, NY, USA: Acm, 2018, pp. 71–81.

[Roj15] Rojas, J. M. et al. "Automated Unit Test Generation during Software Development: A Controlled Experiment and Think-Aloud Observations". *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. Issta 2015. New York, NY, USA: Association for Computing Machinery, 2015, 338–349.

[Rui15] Ruiz, E. et al. *Beyond API Signatures: An Empirical Study on Behavioral Backward Incompatibilities of Java Software Libraries*. Tech. rep. Department of Computer Science, University of Texas at San Antonio, 2015.

[Rul18] Rule, A. et al. “Exploration and Explanation in Computational Notebooks”. *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. Chi ’18. New York, NY, USA: Acm, 2018, 32:1–32:12.

[Sal09] Saldaña, J. *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009.

[Sch18] Schermann, G. et al. *Structured Information on State and Evolution of Dockerfiles on GitHub*. Detailed information on the dataset can be found in the paper "Structured Information on State and Evolution of Dockerfiles on GitHub" accepted at the Data Showcase Track of the International Conference on Mining Software Repositories 2018 (MSR 2018). The software used to collect the dataset and instructions on how to use the dataset can be found in the paper’s online appendix: https://github.com/sealuzh/msr18-docker-dataset. 2018.

[Seo14] Seo, H. et al. “Programmers’ Build Errors: A Case Study (at Google)”. *International Conference on Software Engineering (ICSE)*. 2014.

[Sev14] Seven, D. *Knightmare: A DevOps Cautionary Tale*. `https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale/`. 2014.

[Sha48] Shannon, C. E. “A mathematical theory of communication”. *The Bell System Technical Journal* **27**.3 (1948), pp. 379–423.

[Sil12] Sillito, J. et al. “What Makes a Good Code Example?: A Study of Programming Q&A in StackOverflow”. *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*. Icsm ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 25–34.

[Sin99] Sinha, S. et al. “System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow”. *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 1999, pp. 432–441.

[Stö20] Stöckle, P. et al. “Automated Implementation of Windows-related Security-Configuration Guides”. *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020, pp. 598–610.

[Su07] Su, Y.-Y. et al. “AutoBash: Improving Configuration Management with Operating System Causality Analysis”. *SIGOPS Oper. Syst. Rev.* **41**.6 (2007), pp. 237–250.

[Sul16] Sulír, M. & Porubän, J. “A Quantitative Study of Java Software Buildability”. *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. Plateau 2016. New York, NY, USA: Acm, 2016, pp. 17–25.

[Tid18] Tidelift. *Libraries.io*. `https://libraries.io/`. 2018.

[Ton18] Tonder, R. van et al. “Semantic Crash Bucketing”. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Ase 2018. New York, NY, USA: Acm, 2018, pp. 612–622.

[Tre18] Treude, C. & Aniche, M. "Where does Google find API documentation?" *IEEE/ACM 2nd International Workshop on API Usage and Evolution*. Wapi'18. New York, NY, USA: Acm, 2018.

[Url18] Urli, S. et al. "How to Design a Program Repair Bot? Insights from the Repairnator Project". *40th International Conference on Software Engineering, Track Software Engineering in Practice (SEIP)*. Icse 2018. 2018, pp. 1–10.

[Vla18] Vlad, C. `https://pypi.org/`. 2018.

[Wan15] Wang, W. et al. "What is the Gist?: Understanding the Use of Public Gists on GitHub". *Proceedings of the 12th Working Conference on Mining Software Repositories*. Msr '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 314–323.

[Wei13] Weimer, W. et al. "Leveraging program equivalence for adaptive program repair: Models and first results". *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 356–366.

[Wei06] Weimer, W. "Patches As Better Bug Reports". *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. Gpce '06. New York, NY, USA: Acm, 2006, pp. 181–190.

[Wei17] Weiss, A. et al. "Tortoise: Interactive System Configuration Repair". *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. Ase 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 625–636.

[Wu18] Wu, Y. et al. "How do developers utilize source code from stack overflow?" *Empirical Software Engineering* (2018), pp. 1–37.

[Xav17] Xavier, L. et al. "Why do we break APIs? First answers from developers". *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017, pp. 392–396.

[Xin07] Xing, Z. & Stroulia, E. "API-Evolution Support with Diff-CatchUp". *IEEE Transactions on Software Engineering* **33**.12 (2007), pp. 818–836.

[Xu19] Xu, S. et al. "Meditor: Inference and Application of API Migration Edits". *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 2019, pp. 335–346.

[Yan16] Yang, D. et al. "From Query to Usable Code: An Analysis of Stack Overflow Code Snippets". *Proceedings of the 13th International Conference on Mining Software Repositories*. Msr '16. New York, NY, USA: Acm, 2016, pp. 391–402.

[Yan17] Yang, D. et al. "Stack Overflow in Github: Any Snippets There?" *Proceedings of the 14th International Conference on Mining Software Repositories*. Msr '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 280–290.

[Zel02] Zeller, A. & Hildebrandt, R. "Simplifying and Isolating Failure-Inducing Input". *IEEE Trans. Softw. Eng.* **28**.2 (2002), 183–200.

[Zho10] Zhong, H. et al. “Mining API mapping for language migration”. *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 1. 2010, pp. 195–204.