Master Thesis

Title

Humpback: Language Models-Based Code Completion System for Dockerfiles

Supervisor
Prof. Shinji KUSUMOTO

by
Kaisei Hanayama

February 2, 2021

Departmant of Computer Science
Graduate School of Information Science and Technology
Osaka University

Master Thesis

Humpback: Language Models-Based Code Completion System for Dockerfiles

Kaisei Hanayama

Abstract

Server virtualization is broadly used for cost reduction and efficient resource utilization. Containerization, a type of virtualization technology, has become mainstream. Containerization creates multiple virtual servers (i.e., containers) on a single physical server. Each container provides an independent environment, enabling quicker delivery of applications, improved portability, and efficient resource utilization. The object of this study is Docker, the de facto standard containerization platform. Containers in Docker are built by writing configuration files called Dockerfiles. The process of managing infrastructure configuration through machine-readable definition files is called infrastructure as code (IaC). IaC makes it possible to prevent human errors, automate scaling, and apply knowledge gained from conventional software engineering to infrastructure configuration. However, IaC is a relatively new technology field; some domains of IaC have not been thoroughly researched, such as development support, static analysis, and establishing best practices.

This study focuses on code completion, a widely used feature in software development, among unexplored technical areas of IaC. The goal of this study is to construct a code completion system that supports the development of Dockerfiles. The proposed system applies machine learning with long short-term memory to a pre-collected dataset and creates language models, which calculate probability distributions over a sequence of tokens. This system also employs model switching, a solution for a Docker-specific code completion problem. When creating containers in Docker, it is based on image files called base image. The Linux distribution differs depending on the base image, and the contents of the Dockerfile differ accordingly. Thus, model switching is introduced to reflect base image differences; language models for prediction are selected based on the base image. However, the Linux distribution cannot be identified by the base image name in some cases. A base image detector is also created to determine the Linux distribution in such cases.

Humpback, a code completion system for Dockerfile, was implemented to realize the proposal of this study. 21,190 Dockerfiles were collected as the dataset for training and testing. Evaluation experiments were conducted to confirm how accurate Humpback is and verify that model switching improves code completion accuracy. Experiment results show that Humpback has a high average recommendation accuracy of 89.4%. The contribution of model switching to the improvement of its accuracy was also confirmed.

Keywords

Docker
Infrastructure as code
Code completion

Language model
Machine learning
Long short-term memory

Contents

1 Introduction **1**

2 Background **3**

2.1 Code Completion 3

2.2 Language Model 3

2.3 Docker and Infrastructure as Code 4

2.4 Challenges of Infrastructure as Code 4

2.5 Challenges Toward Code Completion in Docker 5

3 Humpback: Code Completion System for Dockerfiles **7**

3.1 System Overview 7

3.2 Methodology 7

3.2.1 Learning Phase 7

3.2.2 Prediction Phase 8

3.3 Data Cleansing 8

3.4 Implementation 9

4 Evaluation Experiment **11**

4.1 Evaluation Metrics 11

4.2 Dataset 11

4.3 Learning 12

4.4 Experiment Design 12

4.5 Experiment Results 13

5 Discussion **19**

5.1 Accuracy Improvement Through Model Switching 19

5.2 Differences by Linux Distribution 19

5.3 Differences by Syntax 19

6 Threats to Validity **21**

6.1 Threats to Internal Validity 21

6.2 Threats to External Validity 21

7 Related Works **22**

7.1 Code Completion 22

7.2 Infrastructure as Code 23

8 Conclusion **24**

Acknowledgements **25**

References **26**

List of Figures

1 Screenshot of Humpback 2
2 Google Trends data for search term "Infrastructure as Code" 5
3 Dockerfile example 6
4 Overview of learning phase 7
5 Data processing flow 8
6 Overview of prediction phase 9
7 Data cleansing examples 10
8 Histogram of token numbers in Dockerfile 12
9 Syntax definition example 14
10 Experiment results for Alpine 16
11 Experiment results for Debian 17
12 Experiment results for Ubuntu 18
13 Token numbers in each syntax and their percentages 20

List of Tables

1 Evaluation metrics example . . . 11
2 Number of Dockerfiles and versions . . . 12
3 Details of learning . . . 13
4 Average scores for experiment results in Docker syntax . . . 15
5 Average scores for experiment results in Shell syntax . . . 15

1 Introduction

Server virtualization is broadly used for cost reduction and efficient resource utilization [1]. Containerization, a type of virtualization technology, has become mainstream [2]. Containerization creates logical compartments (i.e., containers) on the host operating system (OS). Each container provides an independent environment, enabling quicker delivery of applications, improved portability, and efficient resource utilization. Docker[1] is the de facto standard containerization platform [3]; over 87% of information technology companies use Docker [4]. Docker is also widely used in the open-source software community [5].

Containers in Docker are configured by writing imperative instructions in files called Dockerfile. It is called infrastructure as code (IaC) that the process of managing infrastructure configuration through machine-readable definition files [6, 7]. IaC makes it possible for developers to manage infrastructure configuration in the same way as application code, which enables automated scaling, prevents human error, and allows the incorporation of know-how cultivated in conventional software engineering [8–10]. However, IaC is a relatively new technology field, and thus some areas are still unexplored [11], such as development support, static analysis, and establishing best practices.

This study focuses on code completion, a widely used feature in software development [12, 13]. A code completion system for emerging technology, such as Docker, can considerably improve productivity by reducing common errors and reusing existing knowledge.

The contributions of this thesis are as follows:

1. **Challenges towards code completion are outlined.** One concern when building a Docker-specific code completion system is base image differences. A base image, which includes a Linux distribution, is an image file on which a container is created. The contents of Dockerfiles differ considerably depending on the base image. For accurate code completion, base image differences must thus be taken into account (section 2.5).

2. **A solution to Docker-specific challenges is presented.** In this study, the code completion system is realized by treating Dockerfiles' contents as time-series data. Long short-term memory (LSTM) [14] is employed to generate language models. Model switching is introduced to overcome the problem caused by base image differences. With model switching, language models for prediction are selected based on the base image. However, the Linux distribution cannot be identified by the base image name in some cases. A base image detector is also created to determine the Linux distribution of input Dockerfiles (section 3.2).

3. **A Docker-specific code completion system, Humpback, is implemented.** Humpback helps developers to reduce errors and be more efficient when writing Dockerfiles. Figure 1 is a screenshot of Humpback. Humpback is available online and can be used in a web browser.[2] Evaluation experiments were conducted to confirm how accurate Humpback is and verify that model switching improves code completion accuracy. Experiment results show that Humpback has a high average accuracy of 89.4% and is useful for developing Dockerfiles. It is also confirmed that model switching improves prediction accuracy (section 4.5).

[1] https://www.docker.com/

[2] https://sdl.ist.osaka-u.ac.jp/∼k-hanaym/humpback/

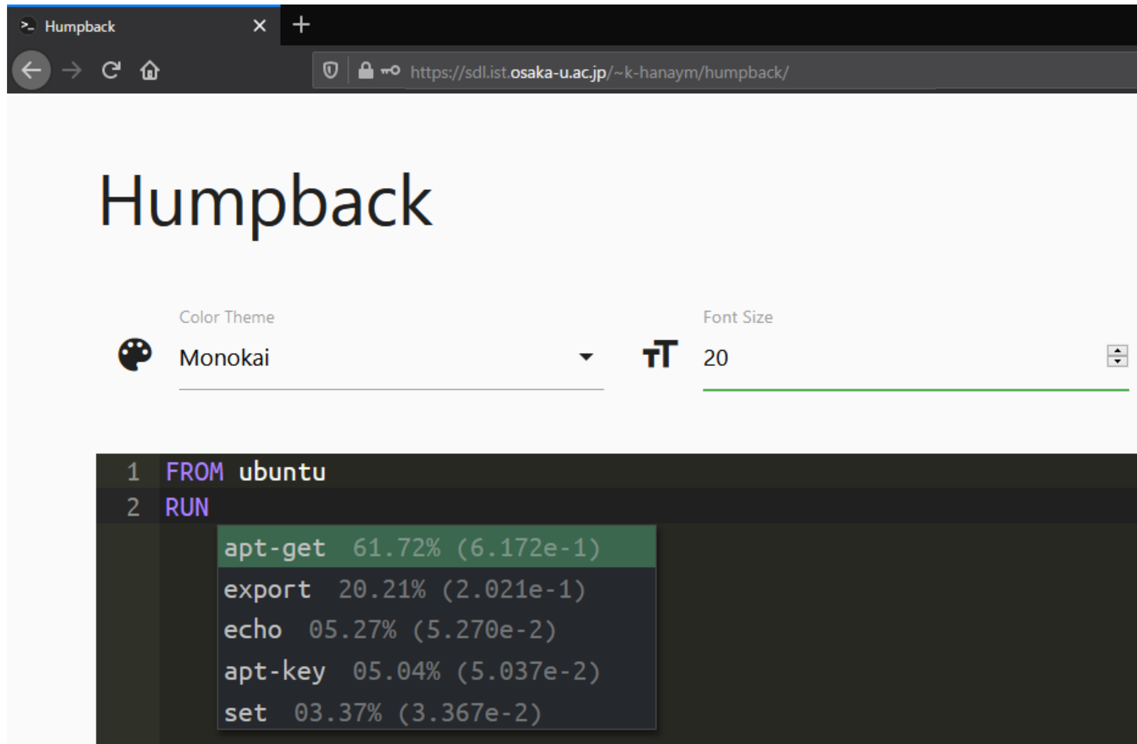


Figure 1: Screenshot of Humpback

The remainder of this thesis is organized as follows: Section 2 shows background knowledge of the pillars of this study. Section 3 presents the proposal code completion system, Humpback. Section 4 explains the experiment and evaluation of Humpback. Section 5 discusses the experiment results. Section 6 reviews the threats to validity of the evaluation. Section 7 provides related works for this study. Finally, Section 8 summarizes this thesis and provide future works.

2 Background

This section provides background knowledge of this study's pillars; code completion, language model, Docker, and IaC. The challenges toward developing a code completion system for Dockerfiles are also introduced.

2.1 Code Completion

Code completion is an extensively used feature in software development [13]; developers use code completion as frequently as several times a minute [12]. A pop-up dialog is used to display a list of candidate words after typing some characters. Developers select the desired word from the list, which reduces typos and other common errors. Another benefit is the facilitation of descriptive (i.e., long) names for variables, methods, and other entities. Manually entering long names is cumbersome and error-prone. These problems can be solved by automating the input process with code completion.

However, traditional code completion systems display all candidate words in alphabetical order without considering the preceding and following contexts. Developers must choose the appropriate one from a too-long list, which means traditional code completion can not contribute much to improving development efficiency.

Therefore, many intelligent code completion systems have been proposed to overcome the problem of traditional ones [15–23]. Systems that use statistical language models such as N-gram [24–26], Transformer [27,28], and recurrent neural network (RNN)-based approaches [29–31] have achieved high performance. Compared to a traditional code completion system, an intelligent one more effectively enhances developer productivity.

2.2 Language Model

A language model is a probability distribution over sequences of words. Given token sequence $S = t_1, t_2, ..., t_m$, the language model assigns the probability P to the whole sequence:

$$P(S) = P(t_1)p(t_2|t_1)p(t_3|t_1t_2)...p(t_m|t_1t_2...t_{m-1}) \quad (1)$$

$$= \prod_{i=1}^{m} p(t_i|t_1t_2...t_{i-1}) \quad (2)$$

Programming languages are languages that contain predictable statistical properties [24]. This probability thus indicates the relative likelihood of words, which allows the construction of code completion systems. Intelligent code completion systems consider the context and calculate probabilities based on language models to narrow the candidate word list.

The probabilities are difficult to calculate when the number of token sequences m is enormous. Therefore, the N-gram model [24] calculates probabilities from the previous $n-1$ tokens:

$$\prod_{i=1}^{m} p(t_i|t_1t_2...t_{i-1}) = \prod_{i=1}^{m} p(t_i|t_{i-(n-1)}t_{i-(n-2)}...t_{i-1}) \quad (3)$$

This calculation is based on n-th order Markov property; the probability of observing the i-th token t_i in the context history of the preceding $i-1$ tokens can be approximated by the probability of observing it in the shortened context history of the preceding $n-1$ tokens.

In recent years, RNN has shown remarkable performance in modeling programming languages with the rise of machine learning [29–31]. RNN can capture longer dependencies than the N-gram model due to the recursion mechanism, which allows for more effective training. LSTM [14] is a common variant of RNN. The vanishing gradient problem in RNN is eased by employing LSTM blocks, powerful gate mechanisms to remember and forget information about the context selectively.

2.3 Docker and Infrastructure as Code

Docker is an open containerization platform for developing, shipping, and running applications [3]. Docker isolates applications from the development environment with containers, allowing quicker delivery of applications, improved portability, and efficient resource utilization. Due to its rapid rise in popularity, Docker has become the de facto standard container technology; over 87% of information technology companies use Docker [4]. Docker is also widely used in the open-source software community [5].

Containers in Docker can be built by interactively executing commands or by creating configuration files called Dockerfiles. Figure 3 is a Dockerfile example. The numbers on the left side in Figure 3 are line numbers. Dockerfile sets up containers through imperative instructions, enabling reproducible builds [32]. The process of specifying the environment in which software systems will be tested and/or deployed through machine-readable definition files is called IaC [6, 7]. Developers can manage infrastructure configuration in the same way as application code, which prevents human error, automates scaling, and allows the application of know-how cultivated in software engineering [8–10]. Interest in IaC has thus grown among developers and researchers [33, 34]. Figure 2 shows Google Trend[3] data related to the search term "Infrastructure as Code". The x-axis presents months, and the y-axis presents the *Interest Over Time* metric[4] determined by Google Trends. This figure shows that interest in IaC has increased steadily after 2015.

2.4 Challenges of Infrastructure as Code

As described in the previous section, interest in IaC is growing steadily, whereas research on IaC is still in its infancy. Rahman et al. [11] stated that the count of IaC-related publications is low compared to that of software engineering. The topics studied in IaC-related publications are divided into four categories: 1) framework or tools for infrastructure as code, 2) adoption of infrastructure as code, 3) empirical study related to infrastructure as code, and 4) testing in infrastructure as code. These topics mainly focus on implementing or extending the practices of IaC itself. Therefore, it can be said that there is room to utilize the knowledge in software engineering to unexplored technical areas of IaC, such as development support, static analysis, and establishing best practices.

[3]https://trends.google.com/trends/explore?q=Infrastructure%20as%20Code

[4]Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term.

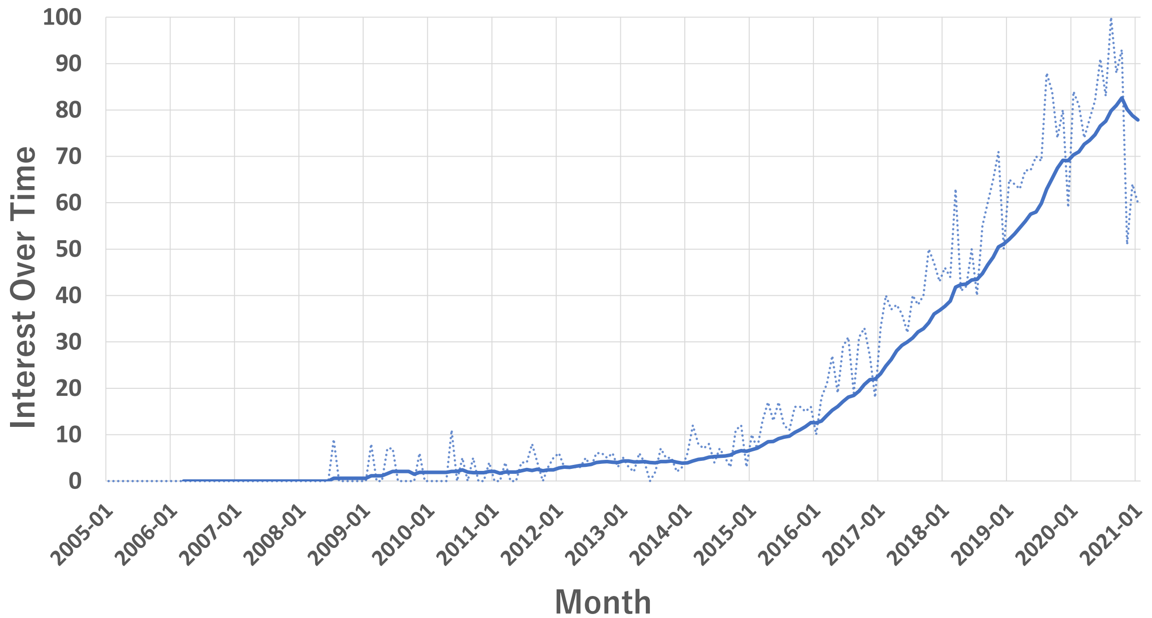


Figure 2: Google Trends data for search term “Infrastructure as Code”

2.5 Challenges Toward Code Completion in Docker

This study focuses on code completion among the under-researched areas described in the previous section. One concern in building Docker-specific code completion systems is the difference in the base image. Base image is the image file on which the container is created, specified by the `FROM` instruction in Dockerfile, like line 3 in Figure 3.

Dockerfile has *nested language*; embedded scripting languages (mainly bash) are described in a nested state in the top-level syntax [2]. The contents of Dockerfiles differ considerably depending on the base image since the Linux distribution is determined here. For example, for an Ubuntu base image, the `apt-get` command is used in the `RUN` instruction, whereas for a CentOS base image, the `dnf` command is used. The majority of the Dockerfiles' contents is this `RUN` instruction (section 5.3). Hence, it is challenging to perform code completion with high accuracy if the base image difference is not considered.

```
1: # create container
2: # from CentOS base image
3: FROM centos:centos8
4:
5: # system update
6: RUN dnf -y update && dnf clean all
7:
8: # install Apache httpd
9: RUN dnf install -y httpd
10:
11: # copy index file
12: ADD ./index.html /var/www/html/
13:
14: # expose port 80
15: EXPOSE 80
16:
17: # display messages
18: RUN echo "now running..."
19:
20: # Start Apache
21: CMD ["/usr/sbin/httpd", "-D"]
```

Figure 3: Dockerfile example

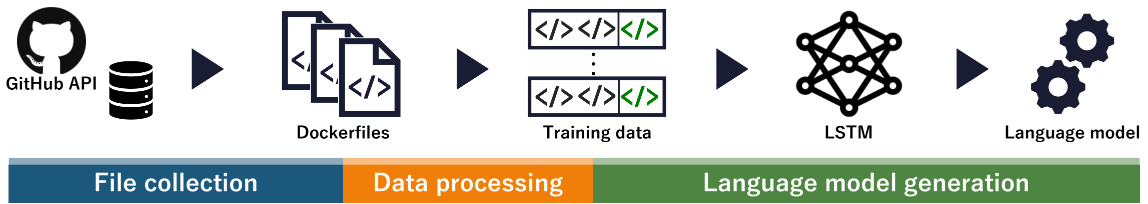


Figure 4: Overview of learning phase

3 Humpback: Code Completion System for Dockerfiles

This section introduces Humpback, a code completion system for Dockerfiles. First, an overview of Humpbackis described. Then, the methodology and implementation of Humpback are explained.

3.1 System Overview

A code completion system for Dockerfiles, Humpback, is implemented in this study. Humpback helps developers to reduce errors and be more efficient when writing Dockerfiles. Various methods have been used to implement intelligent code completion systems. Here, language models are employed. Statistically processing pre-collected Dockerfiles and performing contextual predictions make it possible to reuse existing knowledge. Model switching is also introduced to overcome the problem caused by base image differences.

3.2 Methodology

The methodology of Humpback is divided into the learning phase and the prediction phase.

3.2.1 Learning Phase

The learning phase includes file collection, data processing, and language model generation. Figure 4 is an overview of the learning phase.

File collection: The GitHub GraphQL API[5] allows users to find repositories with specific programming language. Repositories with Dockerfiles are searched using the GitHub GraphQL API, pull these repositories in order of their star count (i.e., popularity), and extract the Dockerfiles.

Data processing: Figure 5 shows data processing flow. First, the contents of the collected Dockerfiles are divided into token sequences. The inputs are paired with the expected outputs. For example, for the statement "`FROM centos RUN dnf`", `centos` is expected after `FROM`, and `RUN` is expected after `FROM centos`. Next, these training data are encoded using integer values for the learner to interpret efficiently. The number of elements in the training data varies. Therefore, 0-padding is performed to obtain fixed-length data finally.

Language model generation: Humpback uses language models for prediction. There are several types of language model, including N-gram [24–26], Transformer [27,28], and RNN [29–31]. The proposed method assumes that the contents of Dockerfiles are time-series data. LSTM [14], a variant of RNN used in deep learning and natural language processing, is employed to generate

[5] https://docs.github.com/en/graphql

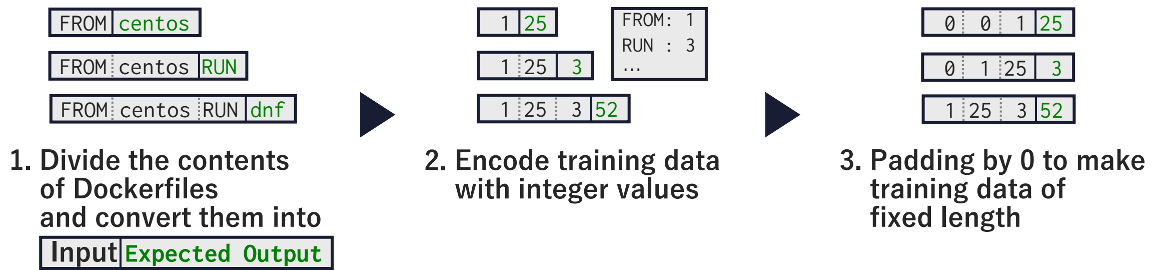


Figure 5: Data processing flow

language models for Humpback. Although a standard (or *vanilla*) RNN can also process time-series data, it is incapable of storing long-term memory. LSTM is an improved version of RNN; it uses individual units, LSTM blocks, in addition to standard units, and can learn long-term dependencies.

3.2.2 Prediction Phase

Humpback uses model switching to overcome the problem caused by base image differences. Figure 6 shows an overview of the prediction phase. Pre-trained language models for each Linux distribution are prepared in advance. Humpback switches models for prediction considering the base image of input Dockerfiles. For instance, if the input Dockerfile's base image is Ubuntu, a model trained with Dockerfiles, whose base images are Ubuntu, is used to make predictions.

However, the Linux distribution cannot be identified from the base image name in some cases. For example, we can guess that "`openjdk:11-jdk`" includes the Java development environment, but we cannot guess its Linux distribution. A base image detector is created to determine the Linux distribution for a given Dockerfile. First, the base image detector builds a container from the input Dockerfile. Then, it identifies the Linux distribution based on the file `/etc/os-release`, which contains OS information. The analyzing results by the base image detector empower Humpback to switch models for prediction even if the base image name does not contain the Linux distribution. For example, the base image detector identified the Linux distribution of `openjdk:11-jdk` as Debian.

3.3 Data Cleansing

Data cleansing was conducted to build a more accurate code completion system. Data cleansing is a process of improving the quality of the data. Duplicates, errors, and shaky notations in a dataset are searched and normalized by deleting or correcting them. Three types of data cleansing are employed in this study: abstraction, embodiment, and denoising. Figure 7 shows examples of each data cleansing.

Abstraction: An example of abstraction is the elimination of tags and digests in the base image. In the `FROM` instruction, tags and digests can be added to specify the version in addition to the base image name. However, there is no significant variation in the commands (e.g., `apt-get`) used by the different versions of the base image. Learning efficiency can thus be improved by absorbing that differences.

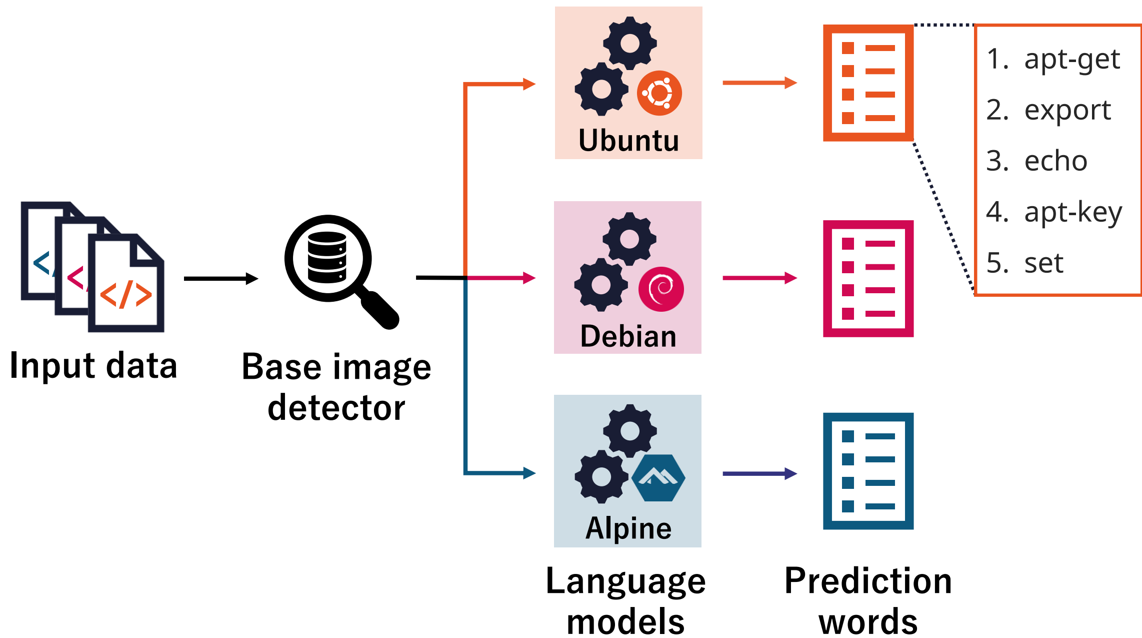


Figure 6: Overview of prediction phase

Embodiment: Replacing `&&` with the previous instruction is an example of the embodiment. Using `&&` in the `RUN` instruction allows multiple commands to be written together in a single instruction. Docker recommends the use of `&&` in the `RUN` instruction since the increase of image size can be reduced [35]. However, training language models is made more accurate by specifying what is indicated by `&&`.

Denoising: Examples of denoising include eliminating meta-information such as author name, version, and build stage alias specification. `LABEL` instruction can be used to set meta-information in Dockerfile. Intermediate images can be named by `AS` instruction. This information is erased as it is considered superfluous for the language model to learn the content of Dockerfile.

3.4 Implementation

Humpback is implemented in Python and Javascript. Three libraries/frameworks are used to train language models, namely TensorFlow[6], a software library for machine learning, Keras[7], a high-level neural network library, and Optuna[8], a hyperparameter auto-optimization framework. As shown in Figure 1, Humpback is available online and ready to use in a web browser. The trained models are deployed using a library called Tensorflow.js[9]. Candidate words are presented immediately, and thus developers can use Humpback without slowing down their development process.

[6]https://www.tensorflow.org/
[7]https://keras.io/
[8]https://preferred.jp/en/projects/optuna/
[9]https://www.tensorflow.org/js

Abstraction

```
FROM ubuntu:18.04
FROM ubuntu@sha256:cbbf2f...
```
▶
```
FROM ubuntu
FROM ubuntu
```

Embodiment

```
RUN apt-get install git
 && apt-get update
```
▶
```
RUN apt-get install git
RUN apt-get update
```

Denoising

```
LABEL maintainer="kaisei<k-hanaym@...>"
FROM centos AS server
```
▶
```
(Deleted)
FROM centos
```

Figure 7: Data cleansing examples

Table 1: Evaluation metrics example

Answer	Recommendations	Rank	Reciprocal rank	Top-1
RUN	**RUN**, FROM, CMD	1	1	✓
apt	yum, apk, **apt**	3	1/3	
install	update, **install**, delete	2	1/2	

4 Evaluation Experiment

This section explains the evaluation experiment to confirm the effectiveness of this study's proposal. First, the experiment schemes, namely evaluation metrics, dataset, and experimental procedure, are described. Then, the experiment results are presented.

4.1 Evaluation Metrics

Evaluation experiments were conducted to see how accurate Humpback is and verify that model switching improves code completion accuracy. Top-k accuracy $Acc(k)$ and mean reciprocal rank MRR [36] were used as metrics for evaluating recommendation accuracy:

$$Acc(k) = \frac{N_{top-k}}{|Q|} \tag{4}$$

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \tag{5}$$

where N_{top-k} is the number of relevant recommendations in the top k suggestions, $|Q|$ is the total number of queries, and $rank_i$ is the rank position of the first relevant word for the i-th query. For both $Acc(k)$ and MRR, a value closer to 1 indicates better model performance. Table 1 shows an example of the evaluation metrics. In this case, $Acc(1)$ is $1/3 \approx 0.33$ and MRR is $(1 + 1/3 + 1/2)/3 = 11/18 \approx 0.61$.

4.2 Dataset

21,190 Dockerfiles were collected using the GitHub GraphQL API. The tokens numbers in each Dockerfile was measured. Figure 8 is the histogram of the token numbers in Dockerfile. The x-axis is the token numbers, and the y-axis is the Dockerfile's frequency. Most Dockerfiles (91.8%) has less than 400 tokens. The minimum token number is 2, the maximum is 1,313, and the average is 223. From this result, we can see that each Dockerfile is not very long.

The dataset contains 6,035 base images. The Linux distribution of each base image was identified by applying the base image detector (section 3.2.2). The numbers of Dockerfiles and their versions for each Linux distribution are shown in Table 2. The major Linux distributions in the dataset are Alpine, Debian, and Ubuntu. The dataset for Ubuntu has the most variety, with 19 versions in 1,497 files. In the table, "Others" includes Amazon Linux, CentOS, Fedora, Oracle Linux Server, and VMware Photon OS.

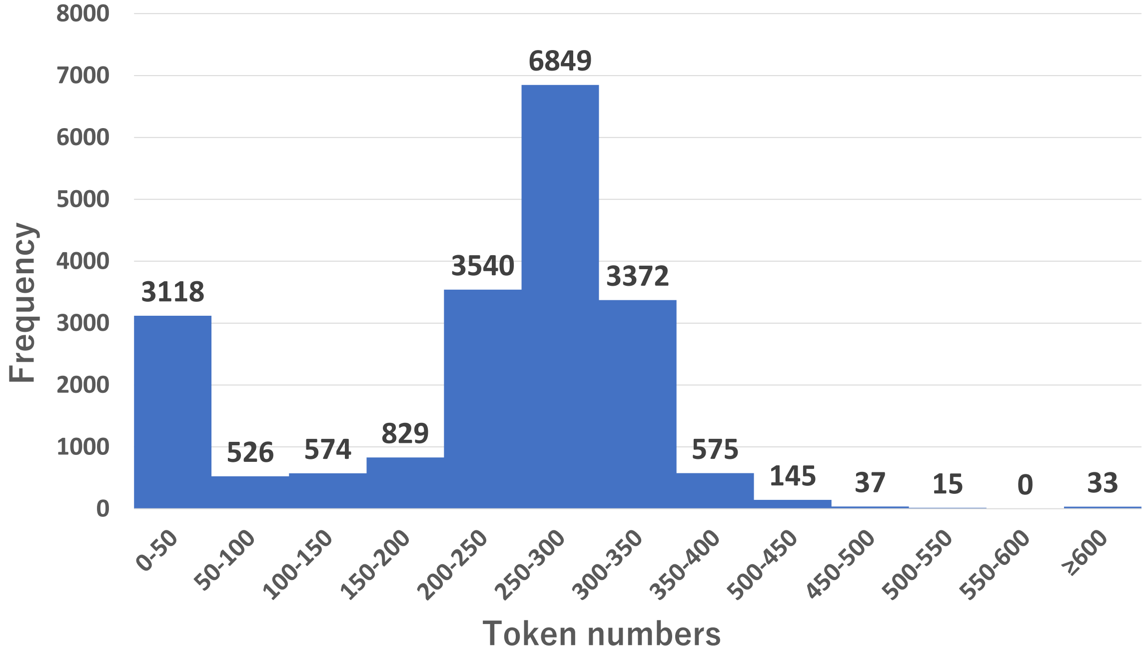


Figure 8: Histogram of token numbers in Dockerfile

Table 2: Number of Dockerfiles and versions

Distribution	# of Dockerfiles	# of versions
Alpine	1,105 (5.2%)	9
Debian	17,011 (80.2%)	6
Ubuntu	1,497 (7.0%)	19
Others	1,577 (7.4%)	-

4.3 Learning

The dataset was divided into training and testing sets (80/20 split on the file level). 4-fold cross-validation was also performed during learning. The details of learning, such as learning duration, activation function, optimizer, and the number of epochs, are shown in Table 3.

There is a large difference in training duration, which is generally related to training data. Debian has the most number of training data, so the most extended training duration. Hyper-parameters such as the activation function, optimizer, and the number of epochs were optimized using Optuna. The activation functions were divided into Sigmoid [37] and Softmax [38]. As the optimizer Adam [39] was chosen for all Linux distributions.

4.4 Experiment Design

The recommendation accuracy was compared for the three major Linux distributions in the dataset, both with and without model switching. For the case without model switching, a generic model trained with all Dockerfileswas produced. Humpback uses multiple language models with model switching, but this generic model used a single language model. All 21,190 Dockerfiles, with-

Table 3: Details of learning

Distribution	Duration	Activation function	Optimizer	# of epochs
Alpine	4h39m	Sigmoid	Adam	56
Debian	1d7h33m	Softmax	Adam	71
Ubuntu	5h12m	Softmax	Adam	24

out their distribution information, were used as training data for the generic model. Two syntaxes were defined in this experiment; descriptions in the `RUN` instruction were defined as *Shell syntax*, and other descriptions as *Docker syntax*. The command part and argument part are distinguished in each syntax (Docker_Command, Docker_Argument, Shell_Command, Shell_Argument). Figure 9 is an example of syntax definition. Docker syntax is colored in blue, and Shell syntax is in red. The shaded part is the command part, and the bold part is the argument part. Therefore, there are three axes of comparison in this experiment: presence or absence of data cleansing and model switching, the Linux distribution, and the syntax.

The experimental procedure are as follows:

1. 100 Dockerfiles are extracted from the dataset.
2. The correct answer is chosen at a random position in each Dockerfile.
3. The contents from the beginning of the file to just before the correct answer (i.e., seeds) are given to the language models.
4. Seeds that are extremely short or whose base image could not be identified are excluded.
5. $Acc(k)$ and MRR are computed by checking the candidate words against the correct answer.

Ten rounds of the above process were performed for each comparison axis. Steps 2. through 4. are explained using the example in Figure 9. For example, "`install`" in the second line is chosen as the correct answer, "`FROM`" in the first line to "`dnf`" in the second line are given to the language models as seeds. If "`centos:centos8`" is chosen as the correct answer, the seeds are too short; the correct answer is randomly chosen again.

4.5 Experiment Results

$Acc(1)$ (Top-1 accuracy) and $Acc(5)$ (Top-5 accuracy) for each Linux distribution are shown in Figure 10, Figure 11, and Figure 12. These figures show the scores for ten rounds of experiments in the form of box plots. Table 4 and Table 5 show the average scores of $Acc(1)$, $Acc(5)$, and MRR in each syntax. "Generic" refers to the generic model (i.e., without model switching). "Humpback" refers to the proposed system Humpback (i.e., with model switching). "Δ" is the degree of improvement by Humpback, i.e., the score by Humpback minus that by generic model. "All" in the distribution column indicates the average score of all Linux distributions. The numbers in bold are the better scores in the same category.

Prediction with Humpback is more accurate for almost all evaluation axes. Humpback achieved a high Top-1 accuracy of 89.4% for the average value of prediction in Docker syntax and Shell syntax (up to 99.2% for Debian with Shell syntax). As described in section 3.4, the candidate words are

```
1: FROM centos:centos8
2: RUN dnf install -y httpd
3: ADD ./test.sh test.sh
4: RUN chmod +r test.sh && bash test.sh
```

Docker-Command / Docker-Argument
Shell-Command / Shell-Argument

Figure 9: Syntax definition example

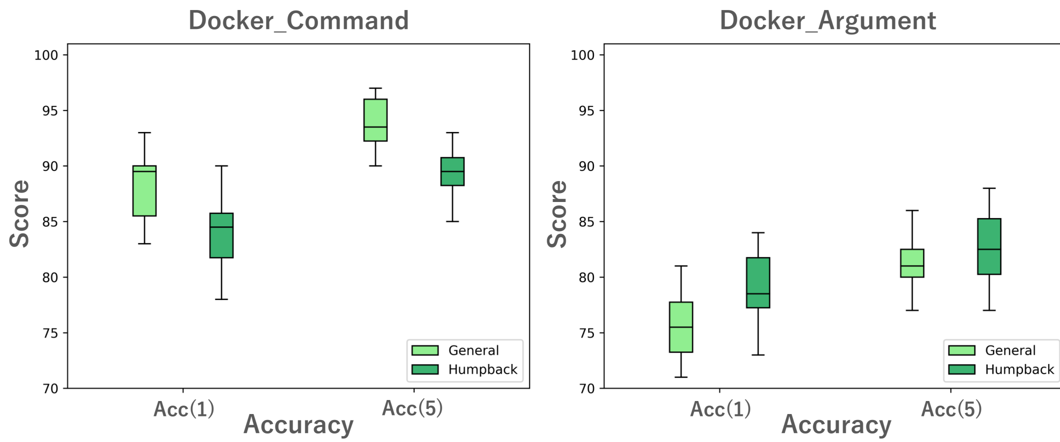
presented instantly. With its quickness and high accuracy, Humpback can significantly improve developers' productivity.

Table 4: Average scores for experiment results in Docker syntax

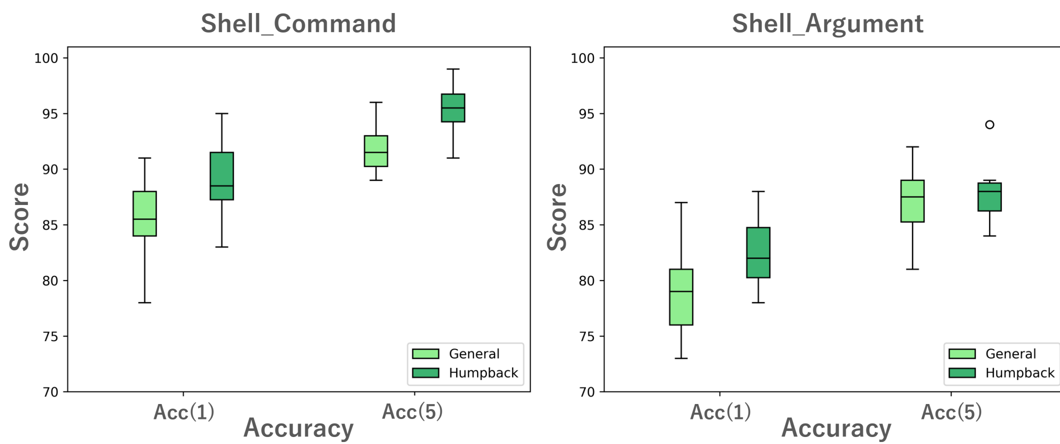
Distribution	**Top-1 accuracy**		**Top-5 accuracy**		**MRR**				
	Generic	Humpback	Δ	Generic	Humpback	Δ	Generic	Humpback	Δ
Alpine	**81.9%**	81.7%	-0.2%	**87.5%**	85.8%	-1.7%	**0.844**	0.837	-0.107
Debian	97.0%	**97.5%**	+0.5%	98.7%	**99.0%**	+0.3%	0.978	**0.983**	+0.005
Ubuntu	74.3%	**87.2%**	+12.9%	82.2%	**90.8%**	+8.6%	0.775	**0.886**	+0.111
All	84.4%	**88.8%**	+4.4%	89.4%	**91.9%**	+2.5%	0.866	**0.902**	+0.036

Table 5: Average scores for experiment results in Shell syntax

Distribution	**Top-1 accuracy**		**Top-5 accuracy**		**MRR**				
	Generic	Humpback	Δ	Generic	Humpback	Δ	Generic	Humpback	Δ
Alpine	82.3%	**85.9%**	+3.6%	89.4%	**91.5%**	+2.1%	0.858	**0.882**	+0.024
Debian	96.5%	**96.8%**	+0.3%	99.1%	**99.2%**	+0.1%	0.977	**0.979**	+0.002
Ubuntu	80.5%	**87.4%**	+6.9%	88.3%	**92.5%**	+4.2%	0.841	**0.898**	+0.057
All	86.5%	**90.0%**	+3.5%	92.3%	**94.4%**	+2.1%	0.892	**0.920**	+0.028

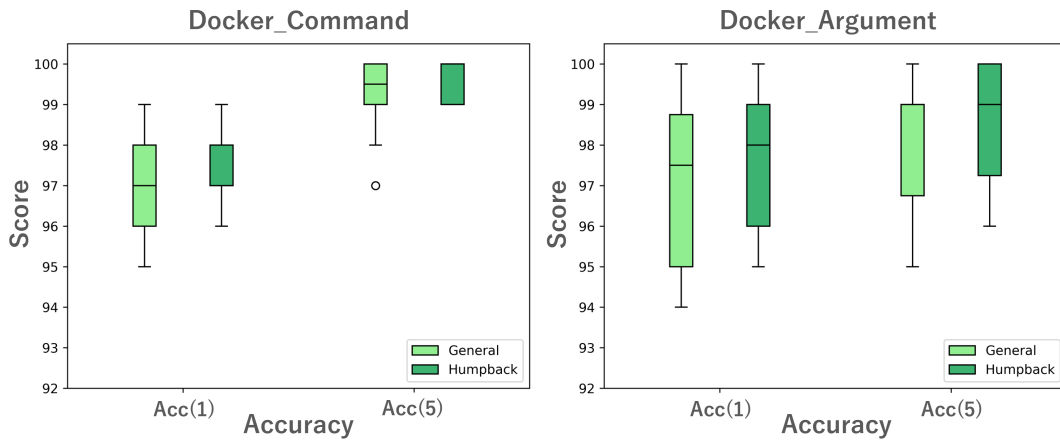


(a) Experiment results for Docker syntax

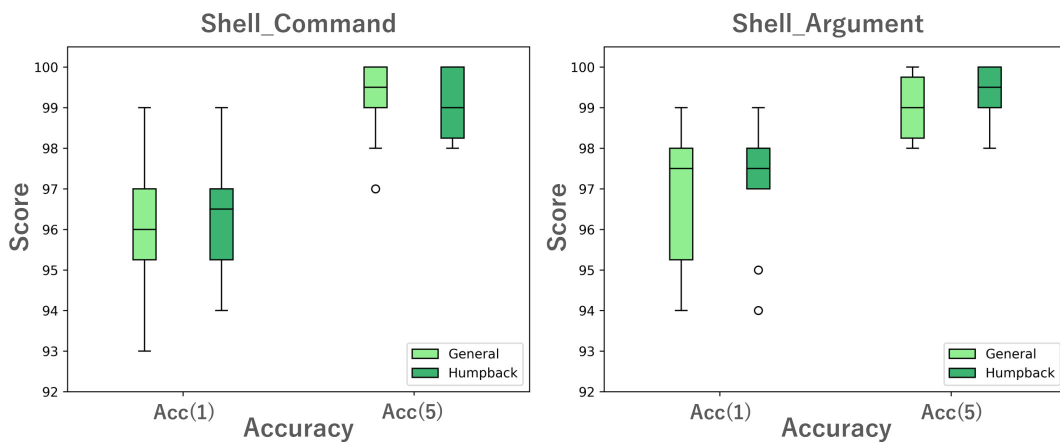


(b) Experiment results for Shell syntax

Figure 10: Experiment results for Alpine

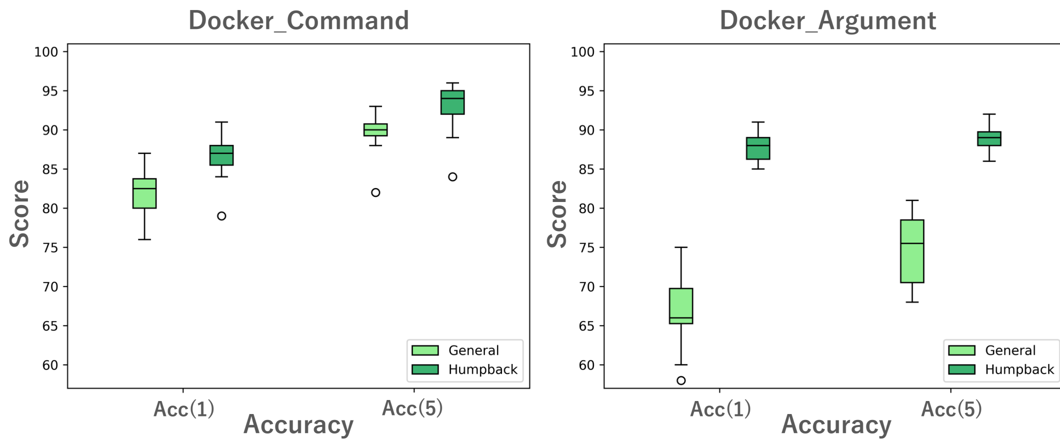


(a) Experiment results for Docker syntax

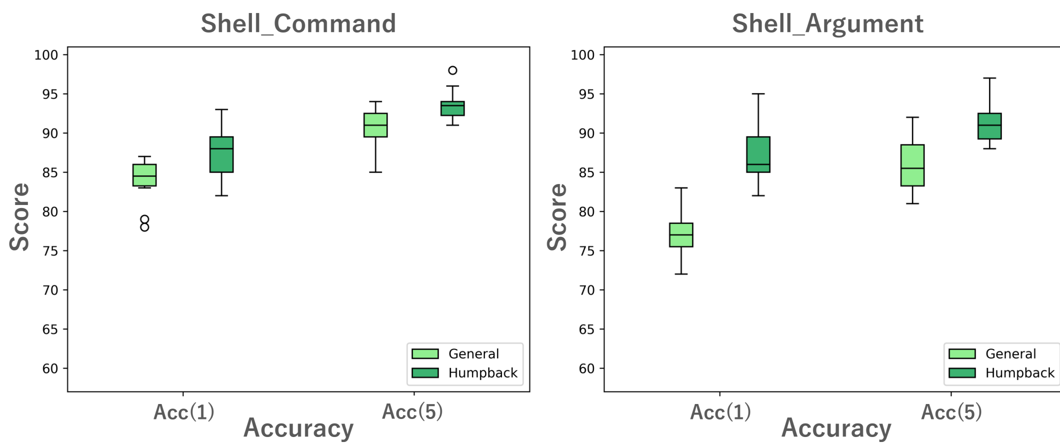


(b) Experiment results for Shell syntax

Figure 11: Experiment results for Debian



(a) Experiment results for Docker syntax



(b) Experiment results for Shell syntax

Figure 12: Experiment results for Ubuntu

5 Discussion

This section discusses experiment results from three viewpoints: model switching, Linux distribution, and syntax.

5.1 Accuracy Improvement Through Model Switching

The experiment results show that model switching is useful for improving accuracy, except for Alpine with Docker syntax. Comparison of the generic model with Humpback reveals that the top-1 accuracy is improved by up to 12.9% (for Ubuntu with Docker syntax). While there are extreme variances in Docker syntax prediction improvement, Shell syntax prediction is improved overall. The accuracy for Shell syntax is improved by 3.5% on average, and there is no case where the introduction of model switching deteriorated the accuracy.

Model switching allows Humpback to reflect the shell command differences between Linux distributions. Therefore, model switching is practical for all distributions in predicting Shell syntax, which leads to improved accuracy. These results show that model switching aids in building a code completion system for Dockerfiles.

5.2 Differences by Linux Distribution

Prediction for Debian is the most accurate for all Linux distributions. One possible reason for this is that there are many training data for Debian. As shown in Table 2, the number of Dockerfiles whose Linux distribution is Debian is 17,011, accounting for about 80% of the total. The more training data makes it possible to train more effectively. Therefore, it can be inferred that the prediction accuracy for Debian, which has the largest number of training data, gets the highest.

Prediction for Ubuntu is more accurate than that for Alpine. However, these distributions have similar numbers of training data (Alpine:1,105, Ubuntu: 1,497). It can be assumed that many of the descriptions in Dockerfiles whose Linux distribution is Ubuntu are similar to each other, whereas fewer similarities in the descriptions in Dockerfiles whose Linux distribution is Alpine. Language models can be trained more efficiently if there are many similar descriptions, even if the training data is almost the same amount. Likewise, if there are many similar descriptions in the training data, there is a high probability that the test data also contain similar descriptions, making prediction easier. For the above reasons, the prediction for Ubuntu can be considered to be more precise than that for Alpine.

5.3 Differences by Syntax

Both the generic model's prediction and one by Humpback show higher prediction accuracy for Shell syntax than that for Docker syntax. The majority of the Dockerfile's content is the Shell syntax that describes the commands to be executed in the container. Figure 13 is a pie chart showing the token numbers in each syntax and their percentages in the dataset. This figure reveals that the Shell syntax accounts for 96.5% of the total Dockerfile's content. The higher the language model's accuracy can be accomplished when the more the training data have similar descriptions, as mentioned in the previous section. Therefore, the efficient training process makes the prediction accuracy for Shell syntax higher than that for Docker syntax.

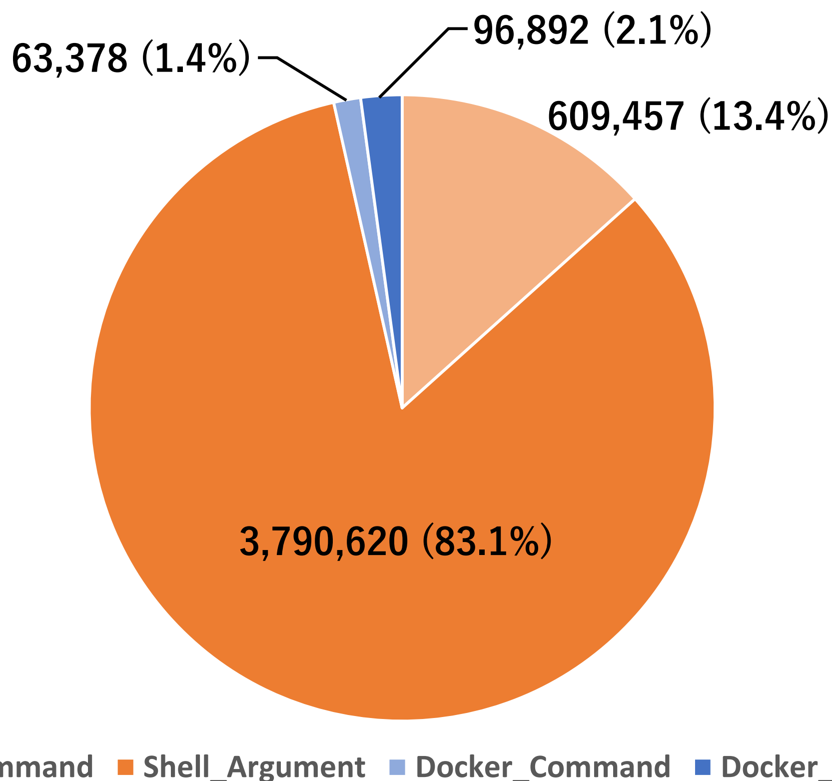


Figure 13: Token numbers in each syntax and their percentages

No trend can be identified for Docker syntax between the command part and the argument part. For Alpine, Docker_Command prediction is more accurate, but for Debian and Ubuntu, Docker_Argument prediction overtakes accuracy. There is also no significant difference between Docker_Command and Shell_Command predictions among all Linux distributions.

However, the Top-5 accuracy of Shell_Command exceeded that of Shell_Argument in all Linux distributions. As shown in Figure 13, the number of Shell_Argument tokens is tremendous, which is about 3.8 million. It can be considered that tokens in Shell_Command have a certain degree of regularity while there is too much variety in Shell_Argument. As a result, the prediction for Shell_Command has become more manageable.

6 Threats to Validity

There are threats to validity in the experiments. They are divided into threats to internal validity and threats to external validity in this section.

6.1 Threats to Internal Validity

Threats to internal validity include the randomness of data generation. Dockerfiles for test data were randomly extracted from the dataset. The correct answer was also randomly generated. Ten rounds of experiments were conducted to minimize bias, but different test data can affect the experiment results. The way of syntax definition is also internal validity. Shell syntax and Docker syntax were defined in this study to analyze the experiment results in detail. This study concludes that there is an improvement in code completion accuracy for both syntaxes. However, changing the syntax definition can yield different results.

6.2 Threats to External Validity

Threats to external validity include the smallness of the dataset. The data set used in this study contains only 21,090 Dockerfiles. Dockerfile should be further collected to get more generic experiment results. The other threat to external validity is the lack of comparison with other code completion systems. A more objective evaluation can ensure the external validity of the experiment results.

7 Related Works

This section clarifies this study's position by describing the related works of code completion and IaC.

7.1 Code Completion

Code completion is an important research topic in software engineering, and various studies have been conducted over the years. Early code completion relies on program history [40], code examples [13], and static type information [41]. Since Hindle et al. [24] discussed the statistical properties in source code, language models began to be used to develop code completion systems [25, 26, 31, 42]. N-gram-based models are one of the most commonly used language models. Tu et al. [25] introduced a caching mechanism in the N-gram model to capture source code's localness and better performance than the conventional one. Hellendoorn and Devanbu [26] proposed Nested Cache N-gram. Nested Cache N-gram is an improved N-gram model with unlimited vocabulary, nested scope, dynamism, and locality in source code. Experiment results of code completion showed that their models exceeded existing language models.

With the rise of machine learning, deep learning-based language models have been widely employed to model source code [31, 43]. Liu et al. [30] proposed code completion with plain LSTM. Bhoopchand et al. [29] modified RNN with a sparse pointer mechanism. Li et al. [31] used a pointer mixture network to tackle the out-of-vocabulary (OoV) problem[10]. RNNs and their variants, especially LSTM-based language models, have shown high code completion performance, as presented in the above studies. This study followed suit and proposed Humpback, a code completion system for Dockerfiles using LSTM. A thorough search of the relevant literature yielded no related article; thus, Humpback is the first code completion system for Dockerfiles.

In recent years, different models from RNN have been proposed to allow for long-term dependencies in the sequential data [44]. Transformer [27] is an architecture based on an attention mechanism [45], which replaces recurrent layers with a self-attention mechanism to reduce sequential computation and capture longer-range dependency. Liu et al. [28] employed Transformer-XL [46], an improved version of Transformer, to construct a code completion mechanism. However, these architectures are not used in this study because Dockerfiles do not have vast long-term dependencies. As mentioned in section 4.2, the token numbers in Dockerfiles is not large, averaging 223. It is sufficient to use LSTM in this study since LSTM-based language models use 200 context words on average [44]. This study confirmed that sufficient accuracy is achieved from the evaluation experiment results.

[10]The OoV problem, as known as the *unknown word problem*, limits the performance of language models, especially when there are many unique words in the dataset. In deep learning-based language models, the dimension of the output layer uniquely corresponds to pre-acquired vocabularies. With an extensive vocabulary, the output layer becomes high-dimensional and computationally expensive. The amount of computation is thus usually reduced by restricting the vocabulary to high-frequency words only during training. Other low-frequency words become unknown words, collectively replaced by special symbols such as `UNK`. However, if the candidate word of the code completion is `UNK`, recommending it is no help for developers. Therefore, it is essential to reduce the number of OoV in the implementation of code completion.

7.2 Infrastructure as Code

Chad Fowler introduced the concept of IaC[11]. The phrase "as code" in IaC means applying the know-how in software engineering, such as code review and version control for infrastructure configuration [6, 34]. Developers who want to configure infrastructure write IaC scripts similar to software code with domain-specific language [47]. Organizations widely use commercial IaC tools or platform, such as Docker, Chef[12], Puppet[13], and Ansible[14] [6, 7, 48].

The most frequently studied topic on IaC is framework or tools; Dolstra et al. [49] proposed Charon, a tool for implementing the practice of IaC, such as automated provisioning and deployment of networks of machines from declarative specifications. Weiss et al. [50] offered a tool called Tortoise, which automatically corrects errors in Puppet scripts. Baset et al. [51] introduced a tool called ConfigValidator. This tool validates IaC artifacts, such as Docker images, by writing rules with declarative language to detect misconfigurations. There are also IaC-related studies that focus on empirical studies and testing. Jiang and Adams [7] researched the co-evolution relationship between the IaC files and the other categories of files in a project, such as source codes, test codes, and build scripts. Sharma et al. [33] investigated *smells* (i.e., recurring coding practices that may have negative impacts) in Puppet scripts and proposed 13 implementation- and 11 design-smells. Hummer et al. [52] proposed a framework to enable automated testing of Chef scripts. Hanappi et al. [53] examined Puppet scripts' convergence and introduced an automated testing framework for asserting reliable convergence. Ikeshita et al. [54] introduced a method to reduce test suites for IaC by checking idempotence.

There are several kinds of studies related to IaC, as we have seen in the preceding paragraph. However, the count of publications is low compared to that of software engineering, even though interest in IaC is growing steadily [11]. Most of the limited number of studies on IaC spotlight on implementing or extending the practices of IaC itself. Our study focuses on the integration of IaC and software engineering, which remains an under-explored area. This study aimed at code completion, a frequently studied feature in software engineering, and developed Humpback, a code completion system for Dockerfiles.

[11]https://www.oreilly.com/radar/an-introduction-to-immutable-infrastructure/

[12]https://www.chef.io/products/chef-infra

[13]https://puppet.com/

[14]https://www.ansible.com/

8 Conclusion

This study proposed a code completion system for Dockerfiles based on a language model. A code completion system, Humpback, was implemented to realize this study's proposal. Humpback is available online and can be used in a web browser. The candidate words are presented instantly, allowing developers to use Humpback comfortably without slowing down their development. Model switching was introduced to overcome a Docker-specific problem and improve the prediction accuracy. Evaluation experiments showed that Humpback has a high average top-1 accuracy of 89.4%. It is also confirmed that model switching improved the accuracy of Humpback.

There are mainly three future works:

Further development of improvement methods: Currently, model switching is introduced in the implementation of Humpback, and its contribution to improving accuracy was discussed in section 5.1. However, considering the issues inherent in Dockerfile and investigating the improvement methods can enhance the Humpback's accuracy.

Improvement of the dataset: The dataset used in this study contains 21,190 Dockerfiles. As considered in section 5.2, the prediction for Debian has the highest recommendation accuracy with the largest number of Dockerfiles. Therefore, additional Dockerfiles should be collected to see if training language models with richer data can improve the recommendation accuracy.

Comparison with other code completion systems: Humpback was implemented as a code completion system for Dockerfiles. However, as described in section 4.4, the evaluation experiments were conducted on generic model, which means no comparison with other code completion systems was executed. It is crucial to compare Humpback with other code completion systems from existing research to evaluate performance objectively.

Acknowledgements

Fortunately, I have received tremendous support from many people throughout this study. This study would not have been accomplished without their help.

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Shinji Kusumoto, for his continuous assistance and encouragement. He sometimes motivated me in a fun way and sometimes gave me serious advice, all of which has always been helped me during this study. I am very grateful for all the help and support I have been received from him over the past six long years, first as a course director when I first entered Osaka University, and then as a supervisor after I was assigned to the laboratory.

I would like to show my most significant appreciation to Yoshiki Higo, Associate Professor, for his practical guidance, valuable and helpful suggestions throughout this study. I talked to him not only at regular meetings but also at various other places. I think he was the person I talked to the most in the student room. It was a very meaningful time for me, not only for chatting but also for getting hints for research and gaining new knowledge from him.

My most profound appreciation goes to Shinsuke Matsumoto, Assistant Professor, for his careful and enthusiastic guidance throughout the entire process of this study. He gave me detailed guidance and various insights as an instructor. This study would never have reached completion without his supports. Also, outside of research, he enriched my student life by sharing various interesting topics with me.

I have greatly benefited from Kusumoto Laboratory members for their assistance and cooperation in various ways; I much thank the second-year master's students, Hideaki Azuma, Tetsushi Kuma, Yuya Tomida, and Tasuku Nakagawa, for their constant encouragement and valuable proposal. I want to acknowledge the first-year master's students, Naoto Ichikawa, Ryoko Izuta, Sho Ogino, Akira Fujimoto, and Aoi Maejima, for active discussion and enriching my student life. I thank the fourth-year undergraduate students, Masashi Iriyama, Kanta Kotou, Masayuki Taniguchi, Tomoaki Tsuru, Hiroto Watanabe, for their efforts to maintain the excellent research environment and make members' study life pleasant. I would like to offer my special thanks to Tomoko Kamiya, a clerical assistant, for her constant and meticulous support. Thanks to all the members, I led a fulfilling research life and made many memories.

I thank the professors at the Graduate School of Information Science and Technology, Osaka University, for their help in lectures and exercises leading up to this research. I also thank all of my friends for their cheering.

Finally, I would like to express my heartfelt gratitude to my family. They have warmly supported, encouraged, and watched over me throughout my student life.

References

[1] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, pages 1–13. Association for Computing Machinery, 2016.

[2] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. A dataset of dockerfiles. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 1–5. Association for Computing Machinery, 2020.

[3] Docker. Docker overview. https://docs.docker.com/get-started/overview/.

[4] Portworx. Annual container adoption report, 2019. https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf.

[5] Jurgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An empirical analysis of the docker container ecosystem on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 323–333. Institute of Electrical and Electronics Engineers, 2017.

[6] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 1st edition, 2010.

[7] Yujuan Jiang and Bram Adams. Co-evolution of infrastructure and source code – an empirical study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 45–55. Institute of Electrical and Electronics Engineers, 2015.

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

[9] Joel Scheuner, Philipp Leitner, Jürgen Cito, and Harald Gall. Cloud workbench -– infrastructure-as-code based cloud benchmarking. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 246–253. Institute of Electrical and Electronics Engineers, 2014.

[10] Matej Artac, Tadej Borovssak, Elisabetta Di Nitto, Michele Guerriero, and Damian Andrew Tamburri. Devops: Introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion*, pages 497–498. Institute of Electrical and Electronics Engineers, 2017.

[11] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108:65–77, 2019.

[12] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006.

[13] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software*

Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, pages 213–222. Association for Computing Machinery, 2009.

[14] Felix A. Gers, Jurgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. In *9th International Conference on Artificial Neural Networks: ICANN '99*, pages 850–855. Institution of Engineering and Technology, 1999.

[15] Awny Alnusair, Tian Zhao, and Eric Bodden. Effective api navigation and reuse. In *2010 IEEE International Conference on Information Reuse & Integration*, pages 7–12. Institute of Electrical and Electronics Engineers, 2010.

[16] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. *ACM SIGPLAN Notices*, 48(6):27–38, 2013.

[17] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. *ACM SIGPLAN Notices*, 49(6):419–428, 2014.

[18] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483. Association for Computing Machinery, 2014.

[19] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with bayesian networks. *Transactions on Software Engineering and Methodology*, 25(1):1–31, 2015.

[20] Andrea Renika Dsouza, Di Yang, and Cristina V. Lopes. Collective intelligence for smarter api recommendations in python. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation*, pages 51–60. Institute of Electrical and Electronics Engineers, 2016.

[21] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: Probabilistic model for code. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, volume 6, pages 4311–4323. JMLR.org, 2016.

[22] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747, 2016.

[23] Alexey Svyatkovskiy, Shengyu Fu, Ying Zhao, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2727–2735. Association for Computing Machinery, 2019.

[24] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 837–847. IEEE Press, 2012.

[25] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280. Association for Computing Machinery, 2014.

[26] Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773. Association for Computing Machinery, 2017.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 5999–6009. Curran Associates Inc., 2017.

[28] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. A self-attentional neural architecture for code completion with multi-task learning. In *IEEE International Conference on Program Comprehension*, pages 37–47. Institute of Electrical and Electronics Engineers, 2020.

[29] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. Learning python code suggestion with a sparse pointer network. 2016.

[30] Chang Liu, Xin Wang, Richard Shin, Joseph E Gonzalez, and Dawn Song. Neural code completion. 2016.

[31] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 4159–4165. International Joint Conferences on Artificial Intelligence Organization, 2018.

[32] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. One size does not fit all: An empirical study of containerized continuous deployment workflows. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 295–306. Association for Computing Machinery, 2018.

[33] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories*, pages 189–200. Institute of Electrical and Electronics Engineers, 2016.

[34] Chris Parnin, Eric Helms, Chris Atlee, Harley Boughton, Mark Ghattas, Andy Glover, James Holman, John Micco, Brendan Murphy, Tony Savor, Michael Stumm, Shari Whitaker, and Laurie Williams. The top 10 adages in continuous deployment. *IEEE Software*, 34(3):86–95, 2017.

[35] Docker. Best practices for writing dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

[36] Dragomir R. Radev, Hong Qi, Harris Wu, and Weiguo Fan. Evaluating web-based question answering systems. In *Proceedings of the Third International Conference on Language Resources and Evaluation*, pages 1153–1156. European Language Resources Association, 2002.

[37] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International Workshop on Artificial Neural Networks*, pages 195–201. Springer, 1995.

[38] Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, 1st edition, 2006.

[39] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015 – Conference Track Proceedings*, pages 284–294, 2015.

[40] Romain Robbes and Michele Lanza. How program history can improve code completion. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326. Institute of Electrical and Electronics Engineers, 2008.

[41] Daqing Hou and David M. Pletcher. Towards a better code completion system by api grouping, filtering, and popularity-based ranking. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 26–30. Association for Computing Machinery, 2010.

[42] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542. Association for Computing Machinery, 2013.

[43] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345. Institute of Electrical and Electronics Engineers, 2015.

[44] Urvashi Khandelwal, He He, Peng Qi, and Dan Jurafsky. Sharp nearby, fuzzy far away: How neural language models use contex. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 284–294. Association for Computational Linguistics, 2018.

[45] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015 – Conference Track Proceedings*, 2015.

[46] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *ACL 2019 – 57th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 2978–2988, 2019.

[47] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 416–430. Association for Computing Machinery, 2016.

[48] Carl Boettiger. An introduction to docker for reproducible research. *Operating Systems Review*, 49(1):71–79, 2015.

[49] Eelco Dolstra, Rob Vermaas, and Shea Levy. Charon: Declarative provisioning and deployment. In *2013 1st International Workshop on Release Engineering*, pages 17–20. Institute of Electrical and Electronics Engineers, 2013.

[50] Aaron Weiss, Arjun Guha, and Yuriy Brun. Tortoise: Interactive system configuration repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 625–636. Institute of Electrical and Electronics Engineers, 2017.

[51] Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci. Usable declarative configuration specification and validation for applications, systems, and cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*, pages 29–32. Association for Computing Machinery, 2017.

[52] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Automated testing of chef automation scripts. In *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference*, pages 1–2. Association for Computing Machinery, 2013.

[53] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. Asserting reliable convergence for configuration management scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 328–343. Association for Computing Machinery, 2016.

[54] Katsuhiko Ikeshita, Fuyuki Ishikawa, and Shinichi Honiden. Test suite reduction in idempotence testing of infrastructure as code. *Tests and Proofs*, 10375:98–115, 2017.