# Recommending Base Image for Docker Containers based on Deep Configuration Comprehension

Yinyuan Zhang, Yang Zhang*, Xinjun Mao*, Yiwen Wu, Bo Lin, Shangwen Wang
National University of Defense Technology, China
{yinyuanzhang,yangzhang15,xjmao,wuyiwen14,linbo19,shangwenwang13}@nudt.edu.cn

***Abstract*—Docker containers are being widely used in large-scale industrial environments. In practice, developers must manually specify the base image in the dockerfile in the process of container creation. However, finding the proper base image is a nontrivial task because manually searching is time-consuming and easily leads to the use of unsuitable base images, especially for newcomers. There is still a lack of automatic approaches for recommending related base image for developers through dockerfile configuration. To tackle this problem, this paper makes the first attempt to propose a neural network approach named DCCimagerec which is based on deep configuration comprehension. It aims to use the structural configuration features of dockerfile extracted by AST and path-attention model to recommend potentially suitable base image. The evaluation experiments based on about 83,000 dockerfiles show that DCCimagerec outperforms multiple baselines, improving Precision by 7.5%-67.5%, Recall by 6.2%-106.6%, and F1 by 7.5%-150.2%.**

***Index Terms*—Docker container, Base image, AST**

## I. Introduction

As the de-facto container technology standard, docker has become one of the most popular containerization tools. Docker allows packaging an application with its dependencies and execution environment into a standardized, self-contained unit, which can be rapidly deployed in any environment without dealing with compatibility and dependency issues [4]. The application encapsulated by docker container is distributed in the form of *image*, which is an executable package that includes everything needed to run the application [9]. Particularly, the contents of a docker image are defined by declarations in the *dockerfile* (see Fig. 1) which specifies the required instructions and the order of their execution, following the notion of IaC.

Similar to the concept of inheritance in object-oriented programming, docker images can define the *FROM* instruction to inherit image definitions from another *base image* [15]. The new image will inherit all the attributes and files encapsulated in the base image [13]. In practice, to find a suitable image, developers often need to manually search the base image from docker registries, e.g., Docker Hub. Docker Hub provides the place for sharing and distributing docker images among the docker community. However, it contains 4.9M+ of available images (as of January 2021), which makes the selection of an image a nontrivial task. Furthermore, the current tool support for searching docker images is limited, as Docker Hub only permits indexing images "by name" [5], developers still need to go to the pages of many images and check their detailed descriptions before making the decision. The entire selection process might be challenging, especially for newcomers who lack experience in using docker. What's more, replacing existing image with a more suitable image can reduce the image size and improve the quality of the image by instruction verification. Therefore, as Cito et al. [6] envisioned, developers need a recommender system that analyzes existing dockerfile codes and produces transformations, thereby recommending appropriate base images.

*Yang Zhang and Xinjun Mao are the corresponding authors.

```
1 FROM ubuntu:18.04
2 COPY . /app
3 RUN  make /app
4 CMD  python /app/app.py
```

**Fig. 1:** An example of dockerfile

To fill the research gap, we propose an automatic approach named **DCCimagerec** based on deep configuration comprehension. Inspired by traditional code representation (e.g., code2vec [3]) and dockerfile characterization (e.g., binnacle tool [8]), DCCimagerec firstly parses dockerfile code into nodes by AST and extracts its node paths, then it builds and trains neural network models based on path-attention algorithms. With the trained model, we can generate embedding vectors for input queries and compute the similarity with all the pre-embedded vectors of the dockerfile corpus. As a result, for a given dockerfile function code (i.e., without base image), DCCimagerec can recommend potential base images for it.

To evaluate DCCimagerec, we design two research questions and perform experiments on a large scale of dockerfiles ($\simeq$83,000). The results show that DCCimagerec can achieve a good performance and significantly outperform eight baseline approaches. In summary, our contributions are as follows:

- We introduce the idea of using path-based representation of dockerfile code for the task of base image recommendation. To the best of our knowledge, this is the first study to explore the problem of recommending base images for docker containers.
- We propose a novel approach named *DCCimagerec*, which uses the structural configuration features of dockerfile extracted by AST and path-attention model to recommend potentially suitable base image.
- Based on a large-scale dockerfile dataset, we verify that DCCimagerec outperforms multiple baselines, improving Precision by 7.5%-67.5%, Recall by 6.2%-106.6%, and F1

by 7.5%-150.2%. Ablation study confirms the effectiveness of each component in our proposed approach.

## II. Motivation

Adopting a suitable base image is important to the docker image creation, as different base image selection may have various effects on the quality and efficiency, and size of the resulting image [6], [15]. For instance, Cito et al. [6] suggests that the same application can be run with a different base image to reduce the overall size and preferably also build time. Recently, with the widespread use of docker, how to find a proper docker base image has become a difficult problem for developers. E.g., a developer posts the question "*What is the advantage over using a heavy base image like Ubuntu 14.04 than a lightweight one like Alpine?*" in StackOverflow[1]. Actually, Docker Hub provides the support for searching docker images "by name" [5], i.e., when developers specify a term, it is exploited to only return all images where such term occurs in the name, in the description, or in the name of the user that built the image. However, this current tool support is limited, developers still need to spend a lot of time to distill the proper image from the image list by checking the detailed description on the image page. Thus, base image recommendation techniques are needed to help developers who seek to find a suitable base image in their dockerfile configuration process without arbitrary decisions.

In practice, the code after the *FROM* instruction can indicate the function of a docker image, involving its application runtime requirements and detailed operations. In this work, we call them as dockerfile *function code*. Unlike the keywords used by developers when searching for base images on Docker Hub, the dockerfile function code itself contains a lot of syntactic structure and semantic information, which can help us better extract useful features for the construction of the recommendation system. Thus, how to represent dockerfile function code effectively needs to be addressed first.

As shown in Figure 2, if we consider the whole function code snippet as a token sequence and embedding each single token, it would separately encoder the tokens $\{$*RUN*, *apt*, *add*, *–no-cache*, *–virtual*, ..., *bzip2-dev*$\}$ in order and aggregate them to represent the function code snippet. Obviously, the structural information of function code snippet is missing. To capture more syntactic structure and semantic information before learning, Alon et al. proposed the embedding model of source code, i.e., code2vec [3]. These model demonstrated that representing code with the AST path is effective. Recently, Henkel et al. [8] developed a rule enforcement tool, *binnacle*, to characterize dockerfiles using phased AST parsing and achieved a good characterization effect. These facts motivate us to integrate the AST-based path technique to our approach for the representation of dockerfile function code.

## III. Approach

In this work, we propose an automatic approach to use **D**eep **C**onfiguration **C**omprehension of dockerfile for docker base

[1]https://stackoverflow.com/questions/43509647

```
1   RUN apk add --no-cache --virtual .ruby-builddeps \
2                   autoconf \
7                   gcc \
8                   bzip2 \
9                   bzip2-dev  \
10      && mkdir -p /opt/yarn
```

**Fig. 2:** A snippet of function code in dockerfile

**image rec**ommendation, called DCCimagerec. The overall framework of DCCimagerec is illustrated in Figure 3.

### A. Phase 1: Characterizing Dockerfile With AST Paths

To extract structural information of dockerfile as much as possible, we employ phased parsing to progressively enrich the AST created by an initial top-level parse. Similar to Henkel et al.'s approach [8], the root node in each dockerfile is set as "DOCKER-FILE". Then, we parse each code line in each dockerfile, and extract all commonly used instruction/parameter nodes. Finally, for each dockerfile, we obtain a sequence of AST nodes. Figure 4 gives an example of the process of phased parsing dockerfile function code. In the first phase, all of the instruction nodes are parsed, i.e., *RUN*. Their specific parameter information is wrapped up in string literals as leaf nodes. During the second phase, we enrich the structured representation by parsing the embedded bash. All the parameter leaf nodes are parsed, e.g., "*apt-get install -y anaconda pycharm*"→"*APT-GET-INSTALL*", "*ARG*", "*PACKAGE*". We keep the tail token information of each parameter node because they can retain dockerfile semantic information. For example, *./scripts/custom.sh* will be retained in our parsed AST structure instead of been ignored.

Following previous work [2], we extract syntactic paths between all leaf nodes traversing through their lowest common ancestor. Then we present each path as a sequence of intermediate AST nodes between two leaf nodes. So elements of the whole path can be divided into three main elements, including start-terminal, statement-queue linked by up and down arrows, and end-terminal. As shown in Figure 4, one AST path is:

(None, ARG ↑ APT-GET-UPDATE ↑ DOCKER-RUN ↑ DOCKER-FILE ↓ DOCKER-RUN ↓ APT-GET-INSTALL ↓ ARG, -y)

Thus, dockerfile code can finally be represented by an arbitrary number of such paths without a great loss of information [12]. Different from shallow features, AST paths contain both semantic information in terminal tokens and structural information in non-terminal nodes, which can provide a full view of dockerfile configuration for a better comprehension.

### B. Phase 2: Trainning Path-attention Model

After obtaining the AST paths of each dockerfile, we first use code encoders to embed a function code into a high-dimensional vector $v$. To create a vector representation $z_i$ for each path $z_i = < t_1^i, n_1^i...n_l^i, t_l^i >$, we encoder the structural sequence($n_1^i...n_l^i$) and terminal token(i.e., $t_1^i$ and $t_l^i$) separately.

*1) Sequence encoder:* In detail, for syntactic sequence token $s_i = n_1^i...n_l^i$, we directly encoder the structural sequence as a single vector based on the fact that the sequence type in our corpus is 4,815, and the max $l_i$ is 13. We represent sequence s = $n_1...n_l$ using a learned embedding matrix $E_s$.

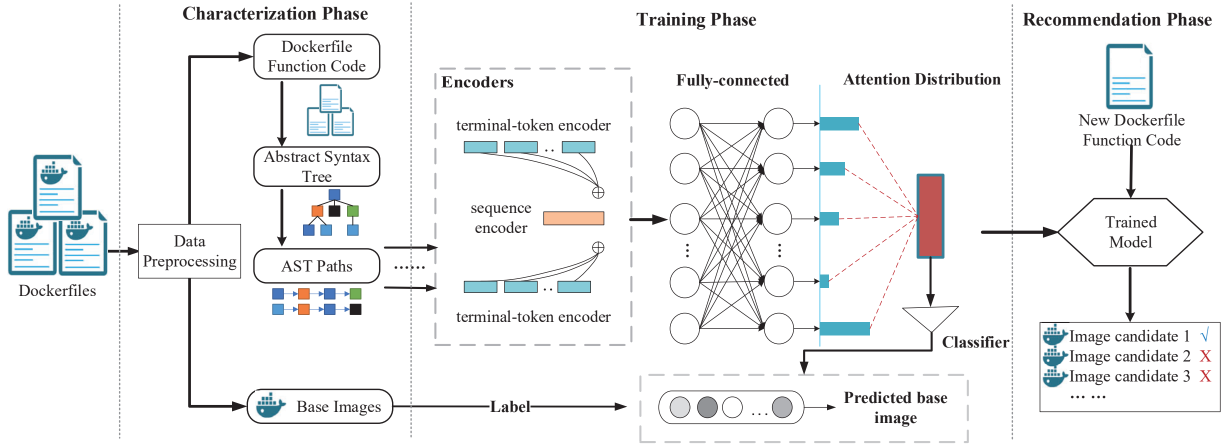


**Fig. 3:** Overall framework of DCCimagerec

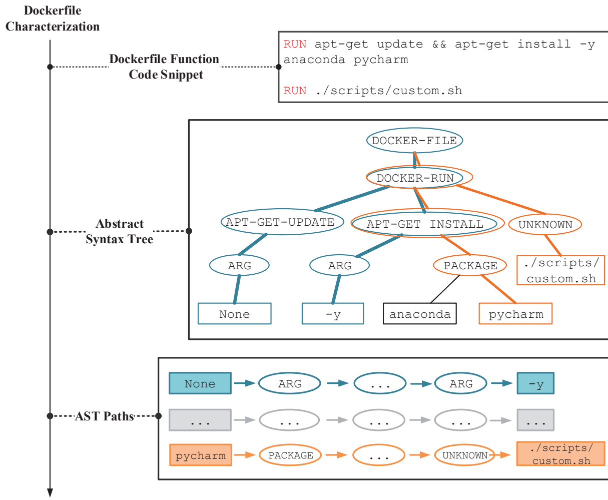


**Fig. 4:** Characterizing dockerfile function code with AST paths

*2) Terminal-token encoder:* If encoding for the whole terminal token directly, some terminal tokens might be too long to capture semantics information. Similar to Allamanis et al. [1], we thus split terminal tokens into subtokens. For example, path '/src/pythonproject' will be split into '/src' and '/pythonproject'. We use a learned embedding matrix $E^{subtoken}$ to represent each subtoken, and then sum the subtoken vectors to represent the full token.

*3) Combined content representation:* For each AST path embedded by three multi dimensional vectors, we concatenate them into a new vector instead of vector summation. The size of embedded vectors of the sequence and the terminal-tokens are both referred as $d$. The concatenated single context vector $z_i \in R^{3d}$, thus, can be described as $[terminal_token(t_1^i), sequence(n_1^i ... n_l^i), terminal_token(t_l^i)]$.

Since every context vector $z_i$ is formed by a concatenation of three independent vectors, a fully connected layer learns to combine its components. The computation of this layer can be described simply as: $\tilde{z}_i = tanh(W \cdot z_i)$. Where $W \in R^{2d \times 3d}$ is a learned weights matrix and tanh is the activation function: tanh(z) = $(e^z - e^{-z})/(e^z + e^{-z})$. The height of the weights matrix W determines the size of $\tilde{z}_i$ , and is set as $2d$ here.

Then we build and train the models of base image recommendation using the path-attention algorithm. Attention mechanism is a model that selects the important paths from the input sequence for each target base image. For example, the target base image *ubuntu* usually aligns with *APT-GET-INSTALL* node in AST sequence. In this task, over these k combined representations $\tilde{z}_1, ..., \tilde{z}_k$, The attention mechanism will work through dynamically path selection. An attention vector $\alpha \in R^{2d}$ is initialized randomly and learned simultaneously with the network. Given the combined context vectors: $\tilde{z}_1, ..., \tilde{z}_k$ The attention weight $\alpha_i$ of each $\tilde{z}_i$ is computed as $\frac{\exp(\tilde{z}_i^T \cdot \alpha)}{\sum_{j=1}^{k} \exp(\tilde{z}_j^T \cdot \alpha)}$. Thus, the aggregated vector, which represents the whole dockerfile snippet lacking *FROM* instruction, is a linear combination of the combined context vectors $\tilde{z}_1, ..., \tilde{z}_k$ factored by their attention weights, i.e., $v = \sum_{i=1}^{k} \alpha_i \cdot \tilde{z}_i$.

### *C. Phase 3: Recommending Potential Proper Base Image*

The path-attention model takes a set of AST paths $z_1, z_2, ..., z_k$ as input and the official base image $y$ as output. $k$ is the number of the sampled length. We formulate the recommendation problem as a multiple classification task, i.e., given a query dockerfile (i.e., function code snippet without base image), our task is to classify it into different labels of base images. In particular, we define the image_tag which is learned as part of training, i.e., $image_tag \in R^{|Y| \times d}$. $Y$ is the set of base_image values found in our dataset corpus, in our experiment is 122. The predicted distribution of the model $q(y)$ is computed as the (softmax-normalized) dot product between the dockerfile vector and each of the image_tag embeddings, i.e., $q(y_i) = \frac{\exp(v^T \cdot image_tag_i)}{\sum_{y_j \in Y} \exp(v^T \cdot image_tag_j)}$.

## IV. Evaluation

To assess DCCimagerec we ask two research questions:

- **RQ1**. How does DCCimagerec perform in the recommendation task of docker base image?
- **RQ2**. Does the specific component affect the performance of DCCimagerec?

Considering that the usage of docker images varies in practice and most docker containers use the official image as their base images [9], so in this study we select those dockerfiles that *FROM* official images as the research object when evaluating the DCCimagerec.

### A. Baselines

Because no existing approach has been proposed on the recommendation problem of docker base image, in this work we compare DCCimagerec with several classical machine learning algorithms (i.e., SVM, Random Forest, and XGBoost) and neural network algorithms (i.e., TextCNN and Bi-LSTM).

Machine learning classifiers usually take feature vectors as inputs. In the field of NLP, embedding techniques such as Doc2vec [11] and Bert [7] have been successfully applied for different semantics-related tasks due to their ability to advance machine learning tasks. Thus, we use Doc2vec and Bert to gain features from each dockerfile as inputs for those machine learning classifiers.

For machine learning baselines, we gain features by embedding (i.e., Doc2vec and Bert) the whole function code for each dockerfile and train classifiers (i.e., SVM, Random Forest, and XGBoost) through the input feature. The vector of function code generated by Doc2vec and Bert is set as default (768 dimensions). For each classifier, we use grid search to optimize important parameters. And we use the "*linear*" kernel in the SVM because it performs better than using "*rbf*" kernel in handling this task. For those neural network baselines, we train function code snippets as plain texts to obtain the token sequences. Similar to Zhang et al. [14], for TextCNN, the kernel size is set to 3 and the number of filters is 100. For Bi-LSTM, the dimension of hidden states is set to 100.

### B. Dataset

In this work, we use the public deduplicated-dockerfile dataset (see Henkel et al. [8]) as our init datasets. This dataset contains 178,506 dockerfiles which were extracted from the public repositories with ten or more stars from January 2007 to June 2019. Then, we remove 53 dockerfiles with only comments but no instructions, resulting in 178,453 dockerfiles. As aforementioned, we select the dockerfiles that use official images for our model training and testing. Thus, we filter dockerfiles that use community docker images and finally get 82,972 dockerfiles using official images.

### C. Evaluation Settings

To measure the recommendation performance of DCCimagerec, we use three evaluation metrics that are most commonly used by previous studies: *Precision*, *Recall*, and *F1-score*. Note that we use the weighted metrics to evaluate the overall performance of different approaches.

As mentioned above, we character Dockerfiles with AST paths. AST paths, including AST node sequence, terminal tokens, thus, will be used as input to the netural model. Following code2vec [3], we encode elements in AST paths with the same 100 dimensions. For AST node sequence, we use 100-dimensional vectors to embedding the whole sequence. For terminal tokens, we randomly sample up to 3 split subtokens from each terminal token, and each subtoken embedding vector is also set as 100 dimensions. AST-based path Encoder, terminal token Encoder, fully-connected, attention, and target base images in our model are jointly trained to minimize the cross entropy. During training, we optimize the parameters of our model using Adam [10] with a batch size of 32, and a learning rate of 0.01. A dropout rate of 0.4 is used to avoid our model overfitting. Although the attention mechanism can aggregate an arbitrary number of inputs, we randomly sampled up to $k$=350 path-contexts from each training example. Because we find that lower values than k=300 shows slightly lower results, and increasing to k>400 does not result in consistent improvement.

To make a fair comparison, each baseline is trained and tested on the same data as our model. Meanwhile, each experiment is calculated from a 10-fold cross validation to evaluate the stability of DCCimagerec. We implement our model using Pytorch and run our experiments on a ubuntu 18.04 server with NVIDIA T4 GPU and 128GB memory.

**TABLE I: Performance of DCCimagerec and baselines**

| Approach | Precision | Recall | F1 |
|---|---|---|---|
| SVM+Doc2vec | 0.503 | 0.475 | 0.420 |
| SVM+Bert | 0.588 | 0.553 | 0.540 |
| RF+Doc2vec | 0.489 | 0.391 | 0.325 |
| RF+Bert | 0.653 | 0.593 | 0.582 |
| XGBoost+Doc2vec | 0.503 | 0.485 | 0.463 |
| XGBoost+Bert | 0.615 | 0.573 | 0.562 |
| TextCNN | 0.762 | 0.761 | 0.756 |
| Bi-LSTM | 0.760 | 0.756 | 0.753 |
| **DCCimagerec** | **0.819** | **0.808** | **0.813** |

## V. Preliminary Results

### A. RQ1: The Effectiveness of DCCimagerec

Table I presents the performance of DCCimagerec and baselines, results presented are averaged from a 10-fold cross validation setup. Among the three traditional classifiers, Random Forest (RF) gains the best performance of 0.653 Precision, 0.593 Recall, and 0.582 F1. However, we find that deep-learning based approaches (i.e., TextCNN and Bi-LSTM) usually obtain better performance than traditional algorithms (i.e., SVM, RF, and XGBoost). This observation makes sense because traditional algorithms mainly rely on the shallow semantic tokens of dockerfile function code, while deep learning-based baseline approaches can capture the important non-linear relationship between dockerfile function code and base image. E.g., the package information in a dockerfile can be captured by the Max-Pooling and sliding window of TextCNN or the memory cell unit in LSTM.

We also find that traditional algorithms with Bert perform better than with Doc2vec. The pre-training corpus of Doc2vec comes from our dataset while Bert's corpus comes from Google. Doc2vec maybe not suitable for parsing dockerfile code since most dockerfiles contain fewer lines of code than traditional code. As seen, DCCimagerec achieves the highest performance among baselines, the improvement of Precision is 7.5%-67.5%, Recall is 6.2%-106.6%, and F1 is 7.5%-150.2%.

### B. RQ2: The Impact of Components

To measure the contribution of the component, we perform an ablation study, i.e., comparing DCCimagerec with six alternative designs based on a 10-folder cross validation:

- *Use Text-based encoder*: We take dockerfile as plain texts, so function code is seen as word token sequence;
- *Use Weakened AST encoder*: We uniformly refer to those unusual commands or parameters as *UNKNOWN*.
- *No sequence encoder*: We remove the non-terminal nodes and only using the first and last terminal nodes in the AST path to represent the code snippet;
- *No terminal-token encoder*: We remove the terminal nodes in the AST path, only using the non-terminal nodes;
- *No terminal-token splitting encoder*: For each node, we did not split it into sub-terminal tokens;
- *No code attention*: We remove the code attention mechanism in the code encoder. The representation vectors of each path are averaged directly without attention weights.

As shown in Table II, DCCimagerec's F1 drops from 0.813 to 0.741 when the model only takes lexical tokens as input. Thus, the contribution of the structural information from non-terminal nodes makes up 9.7%, which demonstrates the importance of the AST-based path encoder for base image recommendation. When using weakened AST encoder, the F1 drops 0.081. It indicates that omitting the detailed information of some commands or parameters will lose a lot of semantic information, which is not conducive to the encoding performance. Furthermore, the experiment results demonstrate the decrement on metrics as the alternative design changes. No terminal-token encoder results in a decrease of F1 (from 0.813 to 0.614). And the performance degradation is greater than no sequence encoder and no terminal-token splitting encoder. This indicates that terminal nodes have better representation performance of dockerfile syntax features than non-terminal nodes. Not using attention mechanism reduces F1 by 7.5%. It shows that not all the paths in a code snippet are useful, and an attentive weighted average can help filter meaningless paths. This result confirms that all the model components positively contribute to the DCCimagerec's performance.

**TABLE II: Performance of different component designs**

| Model Design | Precision | Recall | F1 | $\Delta$ |
|---|---|---|---|---|
| **DCCimagerec** | **0.819** | **0.808** | **0.813** | |
| - use text-based encoder | 0.742 | 0.756 | 0.741 | -0.072 |
| - use weakened AST encoder | 0.781 | 0.715 | 0.732 | -0.081 |
| - no sequence encoder | 0.776 | 0.755 | 0.765 | -0.048 |
| - no terminal-token encoder | 0.693 | 0.573 | 0.614 | -0.199 |
| - no terminal-token splitting encoder | 0.800 | 0.776 | 0.783 | -0.030 |
| - no code attention | 0.779 | 0.744 | 0.752 | -0.061 |

## VI. Conclusion

In this paper, we propose a novel approach DCCimagerec, which effectively recommends potential proper base images for docker containers. DCCimagerec uses AST and path-attention neural network to leverage the syntactic structure and semantic features of dockerfile function code. To evaluate the performance of DCCimagerec, we compare DCCimagerec with eight state-of-the-art baselines. Our experimental results show that DCCimagerec outperforms multiple baselines. We also perform ablation studies and demonstrate that all model components in DCCimagerec can boost the performance.
would like to improve the performance of DCCimagerec by

In future work, we have three research extensions. First, we will extend the evaluation by investigating the impact of query length on DCCimagerec's performance, and qualitatively analyzing the failed recommendation instances. Secondly, we considering more structural and semantic information (e.g., docker image's size and description). Finally, we plan to design and integrate dedicated bots in Dockerfile configuration tools that help developers find suitable base images efficiently.

## Acknowledgment

This work was supported in part by the Program of A New Generation of Artificial Intelligence 2030 (Grant No.2018AAA0102304) and in part by the Laboratory of Software Engineering for Complex Systems.

## References

[1] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning (ICML)*, pages 2123–2132, 2015.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices*, 53(4):404–419, 2018.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. In *Proceedings of the ACM on Programming Languages (POPL)*, pages 1–29. ACM, 2019.

[4] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, 2015.

[5] Antonio Brogi, Davide Neri, and Jacopo Soldani. Dockerfinder: multi-attribute search of docker images. In *International Conference on Cloud Engineering (IC2E)*, pages 273–278. IEEE, 2017.

[6] Jürgen 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 *International Conference on Mining Software Repositories (MSR)*, pages 323–333. IEEE, 2017.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina N Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[8] Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. Learning from, understanding, and supporting devops artifacts for docker. In *International Conference on Software Engineering (ICSE)*, pages 38–49. ACM, 2020.

[9] Md Hasan Ibrahim, Mohammed Sayagh, and Ahmed E Hassan. Too many images on dockerhub! how different are images for the same system? *Empirical Software Engineering*, 25(5):4250–4281, 2020.

[10] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[11] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International Conference on Machine Learning (ICML)*, pages 1188–1196, 2014.

[12] Zhensu Sun, Yan Liu, Chen Yang, and Yu Qian. Pscs: A path-based neural model for semanticcode search. *arXiv preprint arXiv:2008.03042*, 2020.

[13] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. An empirical study of build failures in the docker context. In *International Conference on Mining Software Repositories (MSR)*, pages 76–80. ACM, 2020.

[14] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

[15] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. One size does not fit all: an empirical study of containerized continuous deployment workflows. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 295–306. ACM, 2018.