Exploring the Dependency Network of Docker Containers: Structure, Diversity, and Relationship

Yinyuan Zhang*
National University of Defense Technology, China
yinyuanzhang@nudt.edu.cn

Yang Zhang*
National University of Defense Technology, China
yangzhang15@nudt.edu.cn

Yiwen Wu
National University of Defense Technology, China
wuyiwen14@nudt.edu.cn

Yao Lu
National University of Defense Technology, China
luyao08@nudt.edu.cn

Tao Wang
National University of Defense Technology, China
taowang2005@nudt.edu.cn

Xinjun Mao†
National University of Defense Technology, China
xjmao@nudt.edu.cn

ABSTRACT

Container technologies are being widely used in large scale production cloud environments, of which Docker has become the de-facto industry standard. As a key step, containers need to define their dependent base image, which makes complex dependencies exist in a large number of containers. Prior studies have shown that references between software packages could form technical dependencies, thus forming a dependency network. However, little is known about the details of docker container dependency networks. In this paper, we perform an empirical study on the dependency network of docker containers from more than 120,000 dockerfiles. We construct the container dependency network and analyze its network structure. Further, we focus on the Top-100 dominant containers and investigate their subnetworks, including diversity and relationships. Our findings help to characterize and understand the container dependencies in the docker community and motivate the need for developing container dependency management tools.

CCS CONCEPTS

• **Software and its engineering** → **Software development techniques**; • **Networks** → Network reliability.

KEYWORDS

Docker container, Dependency network, Network analysis

ACM Reference Format:
Yinyuan Zhang*, Yang Zhang*, Yiwen Wu, Yao Lu, Tao Wang, and Xinjun Mao. 2018. Exploring the Dependency Network of Docker Containers: Structure, Diversity, and Relationship. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

*Both are first authors and contributed equally to this work.
†Xinjun Mao is the corresponding author.

1 INTRODUCTION

Docker is one of the most popular containerization tools in current DevOps practice. It enables the encapsulation of software packages into containers and can run on any system [1]. Since its inception in 2013, docker containers have gained 5M+ users and have been downloaded 130B+ times[1]. The contents of a docker container are defined by declarations in the dockerfile [4] which specifies the docker commands and the order of their execution. In practice, every docker container starts from a base image, which has an important effect on the container development and application [23]; it is not uncommon to find containers in docker community depending on other containers, thereby forming complex dependency relationships. Thus, studying docker container dependency is very relevant to the docker-based software development.

On the one hand, with the widespread use and influence of docker, many studies have been recently conducted to investigate the dockerfile usage practices [5, 11, 24]. Those works have emerged a lot of great findings and brought many practical implications to developers, but are not designed to look into the details of docker container dependency. The management practices of container dependency have received little attention, despite being a crucial part of almost all docker projects. On the other hand, prior studies [3, 6] have shown that references between software packages could form technical dependencies, thus forming a dependency network. Recently, network-based analysis of programming language dependency networks has emerged, and many studies have analyzed the topologies of npm, PyPI, CRAN, and Rails [6, 21, 26]. However, there is still a lack of early exploration of the docker container dependency network.

In this work, we take a novel network-based approach for studying dependency networks of docker containers. We use data from more than 120,000 dockerfiles in a public dataset. We compose network of containers based on dependency relations to understand what the static properties and topologies of docker container dependency networks and what the differences and relationships between internal subnetworks.

The goal of this work is to study the state of current dependency network of docker containers, to understand their structure, and to characterize their diversity and coupling relationship. We have formulated the following research themes to guide our research:

[1]https://www.docker.com/company, as of August 2020.

RT1: *What are the characteristics of docker container dependency network?* (see Section 3)

RT2: *How diversified and coupled are the containers in the docker container dependency network?* (see Section 4)

Answers to these questions can help to quantify the state of the docker container ecosystems, give an overview of the trends in container dependency management, and can inform the development of improved dependency management tools. In summary, this work has three core contributions:

- To the best of our knowledge, we are the first to systematically investigate the docker container dependency networks, including their structure, diversity, and relationships.
- We design automatic tools for capturing technical dependencies and constructing dependency networks of docker containers. Data and source codes can be found online[2].
- Our analysis of the container dependency networks result in a series of findings and further distill some implications for different stakeholders.

The rest of our paper is organized as follows. Section 2 discusses the research setup. The detailed results of our study are presented in Section 3 and Section 4. In Section 5, we present the discussion and discuss the related work in Section 6. Finally, we conclude this paper in Section 7.

2 RESEARCH SETUP

To perform our study, we first define the dependency network of docker containers. Then, we present our dataset and tools, and discuss our research questions.

2.1 Dependency Network of Docker Containers

In docker containers, dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image [24]. The instructions stored in dockerfiles define naturally the content of docker containers, including the application, dependencies along with the execution environment. Especially, following the official dockerfile best practice[3], a valid dockerfile must start with a `FROM` instruction where one container inherits infrastructure definitions from another container, e.g., *FROM ubuntu*. The reference relationships captured by `FROM` instruction thus can be regarded as technical dependency between docker containers.

Specifically, we call container that has `FROM` instruction pointing to another container as *Source container*, and container that is pointed to from another container's `FROM` instruction as *Target container*. We say that a pair of containers, C_i and C_j, are a *container pair*, (C_i, C_j), if the `FROM` instruction of container C_i points to C_j. As showed in Listing 1, in the container Tomcat, `FROM` instruction defines the base container as *ubuntu:16.04*, thus can be identified as a container pair (*Tomcat, Ubuntu*), i.e., Tomcat→Ubuntu.

```
FROM ubuntu:16.04

ADD ["startup.sh", "."]

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

Listing 1: An example of Dockerfile of container Tomcat

[2]https://github.com/yinyuanzhang/container-dependency-network
[3]https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices/

Based on those container pairs, we further construct the integral dependency network of docker containers. The *Container Dependency Network*, referenced to Jurgen et al.'s research [3], is defined as a directed network $N = \langle V, E \rangle$. The set of vertices, denoted by V, is all docker containers. The set of edges, denoted by E, indicating the container pairs $E(V) = \{(Ci, Cj) \mid Ci, Cj \in V\}$. Note that we also mark the weight of each edge E_{ij}, which is computed as the number of container pairs (C_i,C_j) created in history. The weight can characterize the frequency of the historical usage of the container pair. For example, if we find three Dockerfiles in our dataset contain the (C_i,C_j), the weight of E_{ij} is 3.

2.2 Dataset and Tools

Our initial dataset comes from public Kaggle dataset[4], which contains 129,463 dockerfiles extracted in early summer 2018. Based on those dockerfiles, we extract all *container pairs* and construct the dependency network, as shown in Figure 1. Specifically, there are five steps:

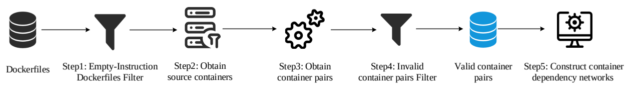
Dockerfiles
Step1: Empty-Instruction Dockerfiles Filter
Step2: Obtain source containers
Step3: Obtain container pairs
Step4: Invalid container pairs Filter
Valid container pairs
Step5: Construct container dependency networks

Figure 1: Overview of our data preprocess.

Step1: Removing dockerfiles with empty FROM instruction. We find that FROM instructions were not correctly defined in many dockerfiles, e.g., missing FROM instruction or base container parameter. To eliminate the threat, we remove those dockerfiles with empty FROM instruction; this yields 128,247 valid dockerfiles.

Step2: Obtaining source containers. In the Kaggle dockerfile dataset, dockerfiles are stored in folders, with the catalog information being *owner/repository/dockerfile*. Thus, we extract the repository name from each dockerfile and find 82,188 unique source containers.

Step3: Obtaining container pairs. Firstly, we capture the base container information by extracting its name from the specification in FROM instruction, i.e., a tuple of the form *namespace/-name(:version)*; This yields 9,270 target container names. Note that we ignore the FROM instruction following with "#" used for interpretative statement. With source containers' information, we finally obtain 130,830 container pairs.

Step4: Filtering out invalid container pairs. We find that in those container pairs, there exist many source containers that equal to their target containers. Also, we find some target containers have no specific meaning, e.g., *"none"*, *"container"*. To reduce threats, We filter out those invalid container pairs. Finally, we collect 124,384 valid container pairs for the dependency network construction.

Step5: Constructing container dependency network. First, we construct the overall dependency network based on all the container pairs. Further, we group container pairs according to their characteristics, and build the subnetworks of selected core containers. Finally, for the overall dependency network and subnetworks, we calculate their network metrics and perform the quantitative study on them.

[4]https://www.kaggle.com/stanfordcompute/dockerfiles/version/1

In order to foster reproducibility and follow-up studies, we provide a comprehensive reproducibility package, containing tool chain, database, queries and analysis scripts. Our dataset and additional scripts can be found online at GitHub[5]. The tool chain consists of separate python projects: dockerfile-dependency[6] is responsible for parsing and generating container pairs, and dockerfile-network[7] is responsible for constructing networks and calculating network metrics.

2.3 Research Question

Our study seeks to answer several research questions that we group into two research themes. The overall motivation is to analyze structure, diversity, and relationships of docker container dependency networks to get insights into current dependency usage and possible issues. Next, we explain the motivation behind each research theme in more detail.

RT1: (Structure) What are the characteristics of docker container dependency network? Currently, not much is known about the static properties and topologies of docker container dependency networks. For example, we know to what extent dependencies are used in traditional packages only [7, 21]. However, we do not know if there are differences in dependency usages across docker containers? In practice, developers can freely define the base image of the container; it is not uncommon to find some containers that are dependent by a large number of other containers, while some are not. However, we do not know which containers are the most influential. Answers to these questions enable us to understand the current state of container dependency networks and would be the starting point for analyzing their internal characteristics. Specifically, we ask the following questions:

RQ1: What does the container dependency network look like?
RQ2: How are container dependencies distributed?
RQ3: What are the dominant containers in the network?

RT2: (Diversity and relationship) How diversified and coupled are the containers in the docker container dependency network? The dominant containers located in the central hub of the network are supposed to be important and could affect a number of other containers. As a result, those core containers and satellite containers form a variety of subnetworks. However, we do not know whether these subnetworks will be diversified or have significant differences. Moreover, due to the interconnected part of the overall network, the connections between the subnetworks are apparent, e.g., the same container in two projects depends on two different containers. However, we do not know much about the closeness / coupling between different subnetworks. Thus, investigation into these questions would help us to understand the diversity and relationship of container dependency networks. Such information could be incorporated in measuring container importance with regards to interconnection in the container dependency network. Specifically, we ask three research questions:

RQ4: What are the types of container dependency subnetworks?
RQ5: What is the difference between subnetworks?
RQ6: How connected are the subnetworks?

[5]https://github.com/yinyuanzhang/container-dependency-network
[6]https://github.com/yinyuanzhang/dockerdependency
[7]https://github.com/yinyuanzhang/dockernetwork

3 RT1: NETWORK STRUCTURE

3.1 RQ1: What does the container dependency network look like?

First, we build the container dependency network based on our dataset and describe its basic network properties.

Approach. As mentioned in Section 2.1, the container dependency network is composed of container pairs. To virtualize the container dependency network, we mainly use the *Gephi tool*[8]. Further, we analyze the basic network structural properties, including *diameter* and *average shortest path*. Specifically, *diameter* is defined as the maximum of the length between any nodes, and *average shortest path* is calculated as the average of all the length between nodes.

Results. The container dependency network consists of 83,030 nodes and 98,248 edges. Figure 2 shows the full network overview, though for visibility we only display nodes with degree of 10 or greater. The size of the node represents the degree of the node, while the thickness of the edge represents the weight of the edge and the color of the edge represents different modular provided by Gephi tool. As visible on the graph, most of the nodes depend on each other which forms huge components, but there are also some isolated nodes that are connected to only a small number of nodes. Furthermore, we find that the size of some nodes or some edges are larger than others, indicating that there exist some dominant nodes in the network, and that some container pairs have been frequently used in history.

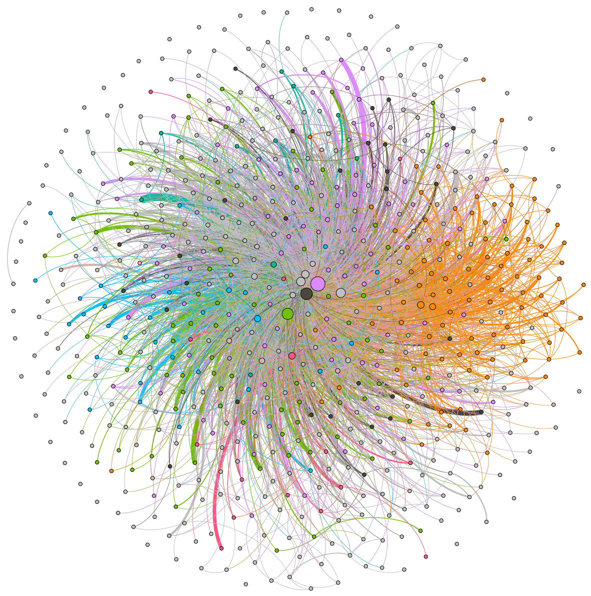
Figure 2: Overview of the docker container dependency network. Redrawn by Fruchterman Reingold algorithm.

The *diameter* computed in the container dependency network is 23 and the *average shortest path* is 5.72. These numbers are larger than the findings reported for developer social networks [25] and project relationship networks [20], implying that container dependency networks are actually looser than other real networks. The

[8]https://gephi.org/

potential reason may be caused by the way the container establishes technical dependencies. In fact, 97.85% dockerfiles in our dataset only contain one FROM instruction, most of source containers only have one target container. A small part of the dockerfiles depend on multiple base images at the same time, which may be related to their design to combine multiple microservices. The configuration characteristic of dockerfile may make technical dependency reliable, but it would cause the overall connectivity of the container dependency network to be very loose, to be further explored in future work.

Answer to RQ1: There are dominant containers in the container dependency network and some container pairs are widely used. However, the interconnection of the docker container dependency network is relatively loose.

3.2 RQ2: How are container dependencies distributed?

Next, we aim to analyze the distribution of container dependencies, i.e., their node degree, among all container nodes.

Approach. For each container node in the container dependency network, we calculate its *indegree*, *outdegree*, and *total degree*. The *indegree* is computed as the number of all the nodes point directly to the given node. The *outdegree* is computed as the number of all the nodes that the given node points to. The total *degree* is the sum of *indegree* and *outdegree*. The degree distribution represents the relative frequency of each value of the node degree in the container dependency network.

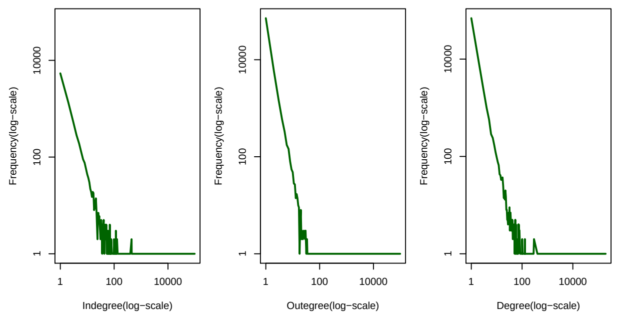

Figure 3: Docker containers' degree distribution. y-axis shows the number of containers having given node degrees.

Results. As shown in Figure 3, we can observe that all distributions display the power law relationship and follow a long tail distribution [2], indicating that container dependency network is a scale-free network. A very small number of containers, however, may occupy a greater degree in the total network. The average node degree of the container dependency network is 2.36 (indegree: 1.18, outdegree: 1.18), indicating that each container may depend on 1.18 other containers and be dependent on 1.18 times. This is consistent with the notion that most containers only depend on only one base image. It also reveals that most containers rely on several central containers. Because using popular docker containers with strong convince is in line with most practitioners, especially for new docker users.

In this network, a successful container may attract more containers to reference, while many containers will stagnate after a short while. The number of dependents of a container may be an important positive feedback factor in determining the influence of a container. However, we find that some containers depend on multiple base images. These containers are also linchpin nodes in the community network, who link containers together into clusters. Thus, for those containers that are at the tail but linking different clusters may also play a huge and important role in the docker container development.

Answer to RQ2: The degree distribution of docker containers follows a typical long-tail distribution. Containers with a large number of dependents or linking different clusters may be important factors that affect the container community.

3.3 RQ3: What are the dominant containers in the network?

In this RQ, we want to identify the most influential (i.e., dominant) containers from the container dependency network.

Approach. We consider dominant containers are those containers with high degrees, because they have more connections in the container dependency network. For each container, we compute its degree values and degree rate (node degrees/all degrees). Then, we range all containers in terms of the degree rate. Since we are most interested in studying the dominant containers, we focus our analysis on the Top-100 containers with the highest degrees.

Results. We find that although Top-100 containers only occupy 0.12% of all containers, they occupy 39.90% of all degrees. Undoubtedly, this indicates the influence of these 100 containers to other containers in the whole network. However, we find that the *indegree* is much greater than the *outdegree* of the same containers. It reveals that these containers are always in the center of the clusters, surrounded by satellite containers. And they only dependent on a few basic images. For example, container *Ubuntu* has 14,975 dependents while it only depends on 11 other containers.

Table 1 gives the Top-20 containers identified in our study. The Top-1 container is *Ubuntu* (occupies 7.63% degrees), followed by *alpine* (4.23%) and *node* (3.76%). We find that in the Top-10 containers, there are strong representation of operating systems (i.e., Ubuntu, alpine, debian, and centos) and programming languages (i.e., python, ruby, golang, and php). Also, other dominant containers belong to the web service (e.g., nginx) or tools (e.g., scratch). Those containers are essential infrastructure for software application and development, so it is very common to find a large number of containers that depend on them. Moreover, those dominant containers and satellite containers could form a variety of subnetworks. What are the specific types of these subnetworks and what are the differences between them? Those questions motivate us to step forward to the RT2.

Answer to RQ3: The indgree of containers plays an important role in measuring the network influence. In the dominant containers list, there are strong representation of operating systems and programming languages.

Table 1: TOP-20 most influential containers.

Container	Indegree	Outdegree	Degree	Degree Rate
ubuntu	14,975	11	14,986	7.63%
alpine	8,302	5	8,307	4.23%
node	7,367	29	7,396	3.76%
debian	6,531	6	6,537	3.33%
python	5,883	26	5,909	3.01%
ruby	3,975	30	4,005	2.04%
centos	3,688	4	3,692	1.88%
golang	3,293	19	3,312	1.69%
php	2,418	33	2,451	1.25%
nginx	1,970	54	2,024	1.03%
java	1,861	34	1,895	0.96%
openjdk	1,685	11	1,696	0.86%
basecontainer	1,556	9	1,565	0.80%
base	951	34	985	0.50%
fedora	745	2	747	0.38%
alpine-node	720	6	726	0.37%
scratch	681	4	685	0.35%
maven	610	12	622	0.32%
tomcat	445	22	467	0.24%
buildpack-deps	449	3	452	0.23%

4 RT2: DIVERSITY AND RELATIONSHIP

4.1 RQ4: What are the types of container dependency subnetworks?

First, we aim to construct the subnetworks of dominant containers and investigate their characteristics.

Approach. To address this RQ, our approach includes the following detailed steps:

1) Constructing subnetworks of dominant containers: Based on the Top-100 dominant containers found in RQ3, we construct their subnetworks. Our operationalization, then, of the container X's subnetwork includes, besides X itself, two types of satellite containers: (a) Type A containers: all containers A depend on X, i.e., $A \longrightarrow X$; and (b) Type B containers: all containers B in which X depends on, i.e., $X \longrightarrow B$. Thus, for each container X, we define its subnetwork as a directed and weighted graph $SubN_X = \langle V, E, W \rangle$, where $V \in \{X, A, B\}$. The weight of each edge is the count of dependency reference for the container pairs.

2) Classifying subnetwork type: In our study, we use the type of core container to represent the type of the entire subnetwork. Inspired by the practical container tags on Docker Hub and the findings of Jürgen et al. [5], we manually categorize the core containers into 5 types, including "*Operating System (OS)*", "*Language Runtime (LR)*", "*Tools or Services (TS)*", "*Application Framework (AF)*", and "*Other*". Specifically, *OS* are containers that manage computer hardware, software resources, and provide common services for computer programs. *LR* are containers that contain a runtime needed to execute applications written in a specific programming language. *TS* are containers that belong to database, DevOps tool, or web server. *AF* are base application infrastructure or frameworks that provide platform for tools and applications. While *Other* denotes containers that do not fit cleanly into any of the above categories.

To verify the validity of our categories, two of our authors were asked to classify the 100 containers separately at the same time. We measure the inter-rater agreement for the first author's categorization and the additional coder (second author); the Fleiss's Kappa value among the three coders was 0.91, which can be considered an almost perfect agreement [12].

3) Computing the subnetwork metrics: To characterize the structure of subnetworks, we use metrics involving *number of nodes*, *number of edges*, and *sum of edge weights* to evaluate the scale of subnetworks and use the metric *interconnectedness* to evaluate the network compactness. Higher interconnectivity means that there are frequent dependencies between nodes in the subnetwork.

$$interconnectedness = \sum Weights / \#Nodes \tag{1}$$

Note that we do NOT use the *diameter* and *average shortest path* metrics to evaluate the compactness of subnetworks. Because during our analysis, we find that most subnetworks are being formed around a central container with the other containers in the subnetwork mostly depending on that central container, which results in a star pattern. Therefore, the diameter and average shortest path in most subnetworks are the same (i.e., 1 and 2).

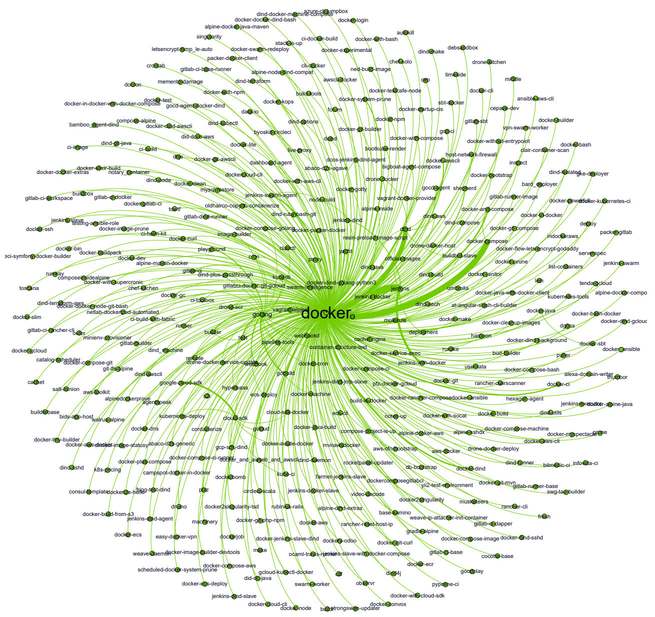

Figure 4: An example of subnetwork: docker.

Results. Figure 4 gives the subnetwork we constructed for the container *docker*[9]. Obviously, core container *docker* occupies the center of this subnetwork. We also find that some parts of satellite containers also connected with each other. Further, for the Top-100 dominant containers found in RQ3, we construct their subnetworks and classify them into 5 categories. In total, 42 out of 100 subnetworks belong to the *TS* category, 21 subnetworks belong to the *OS* category, 18 subnetworks belong to the *LR* category, and 11 subnetworks belong to the *AF* category. Thus, tools and services are the most common type in the container subnetworks, followed by the operating systems, language runtime, and application framework.

[9]https://hub.docker.com/_/docker

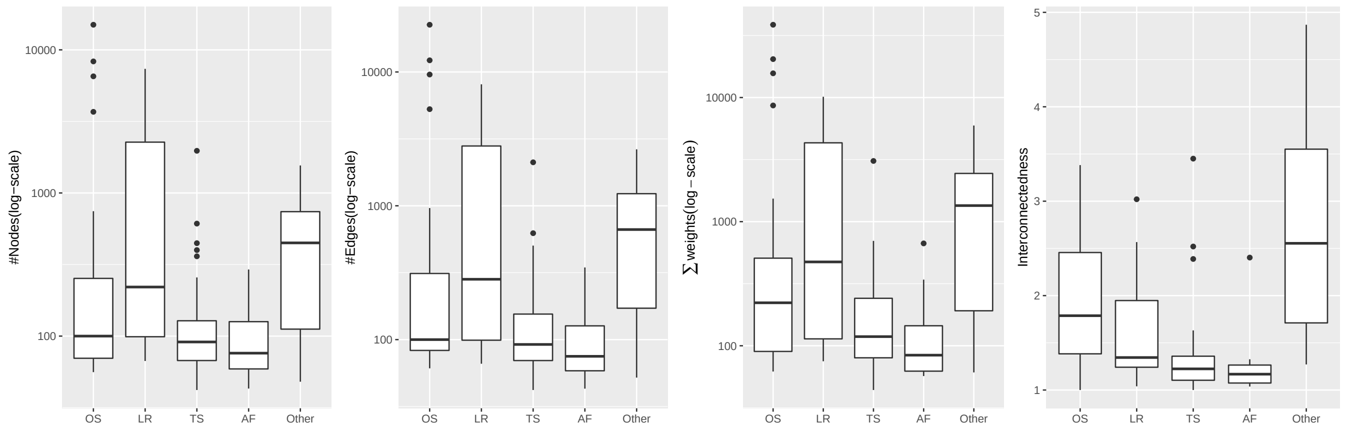

Figure 5: Comparison results of network metrics between subnetworks.

Table 2: Top-20 subnetworks ranked by interconnectedness.

Subnetwork	Type	#Nodes	#Edges	$\sum$weights	Interconnectedness
buildpack-deps	Other	450	852	2,191	4.869
basecontainer	Other	1,557	2,642	5,950	3.821
base	Other	952	1,720	3,295	3.461
apache	TS	71	93	245	3.451
centos7	OS	68	93	230	3.382
scratch	Other	681	1,102	2,214	3.251
alpine-glibc	OS	165	240	508	3.079
alpine-java	LR	194	275	586	3.021
opensuse	OS	100	144	299	2.990
base-alpine	OS	70	88	198	2.829
ubuntu	OS	14,976	22,535	38,577	2.576
openjdk	LR	1,686	2,437	4,328	2.567
composer	TS	125	183	315	2.520
alpine	OS	8,303	12,222	20,395	2.456
cuda	AF	278	346	668	2.403
debian	OS	6,532	9,571	15,671	2.399
openresty	TS	67	72	160	2.388
centos	OS	3,689	5,273	8,630	2.339
alpine-oraclejdk8	LR	126	150	261	2.071
fedora	OS	746	962	1,533	2.055

Table 2 presents the detailed information of the Top-20 subnetworks ranked by the interconnectedness values. We find that container *buildpack-deps* has the most interconnected network structure. *buildpack-deps*[10] is a collection of common build dependencies used for installing various modules, e.g., gems. Interestingly, we find that the subnetworks of *base* and *basecontainer* also have high interconnectedness. But they don't belong to a clear container application category. In the docker project *headspinio/containerbase*[11], the container was used to pre-install the apt package, while in the docker project *carrickpark/containerbase*[12], the container defines the base image of the *carrickpark* application. In addition, in some dockerfiles, we find that the developer did not directly define the base image, but defined it by assigning a value to the "*base*" variable.

[10]https://hub.docker.com/_/buildpack-deps
[11]https://hub.docker.com/r/headspinio/containerbase
[12]https://hub.docker.com/r/carrickpark/containerbase

In general, the network structure and internal connectivity vary between subnetworks. Some small subnetworks (e.g., *apache*) have relatively higher interconnectedness, while some big subnetworks (e.g., *centos*) have relatively lower interconnectedness.

Answer to RQ4: Tools and services are the most common type in the container subnetworks, followed by the operating systems, language runtime, and application framework. However, the network structure and internal connectivity of different subnetworks are diverse.

4.2 RQ5: What is the difference between subnetworks?

Next, we aim to investigate whether and to what extent do subnetworks differ from each other.

Approach. For the network metrics (i.e., #nodes, #edges, $\sum$weights, and interconnectedness), we use statistical methods to compare the differences between subnetworks. We mainly use Wilcoxon test [8], which is a non-parametric test, to compare the difference between two distributions. We also compute Cliff's delta [15], to measure the effect size that quantifies the difference between two distributions. Where the magnitude is assessed using the thresholds, i.e., $|\delta|<0.147$ "negligible", $|\delta|<0.33$ "small", $|\delta|<0.474$ "medium", otherwise "large".

Results. Figure 5 shows the comparison boxplots between different types of subnetworks. For the basic scale metrics, we find that different types of subnetworks may vary. The detailed hypothesis test results are shown in Table 3. Especially, the subnetworks of *LR* have more nodes, more edges, and larger weights. Their network scales are significantly larger than the subnetworks of *TS* and *AF* (p-value<0.05; effect size is large). This is as expected, since *LR* being essential basic conditions for realizing software functions will have a wider range of usage than *AF* or *TS* biased towards implementing a specific function or application in a specific field. E.g., container *Redis* is used as a database and container *Tomcat* is used as a web server. Interestingly, we find there are several outliers in the boxplots of *OS* subnetworks. We manually check and find that those abnormal containers are extremely popular even compared

Table 3: Hypothesis test results between different groups.

	#Nodes			#Edges			∑Weights		
Group	p-value	$\|\delta\|$	effect	p-value	$\|\delta\|$	effect	p-value	$\|\delta\|$	effect
LR-AF	0.011	0.566	***	0.008	0.586	***	0.004	0.626	***
LR-TS	0.003	0.488	***	0.002	0.504	***	0.002	0.509	***
LR-OS	0.143	0.278	*	0.291	0.201	*	0.364	0.175	*
AF-TS	0.511	0.132	-	0.490	0.139	-	0.334	0.193	*
AF-OS	0.226	0.268	*	0.088	0.377	**	0.340	0.463	**
TS-OS	0.374	0.139	-	0.120	0.243	*	0.067	0.287	*

'***' large, '**' medium, '*' small, '-' negligible

with other *OS* containers, e.g., *Ubuntu* and *alpine*. However, the scale differences between *OS* and *LR* are not significant.

As for the interconnectedness, we find that *OS* subnetworks have larger values than other types of subnetworks. Specifically, the difference between *OS* and *LR* is medium (p-value<0.05; Cliff's delta=0.36), the difference between *OS* and *AF* (p-value<0.05; Cliff's delta=0.55), *OS* and *TS* (p-value<0.05; Cliff's delta=0.51), are large. This indicates that containers in the *OS* subnetworks have more connectivity. *OS* only provides a basic operating environment, applications and tools may need to depend on each other and be combined to achieve complete container functions.

Answer to RQ5: There exist some differences in the network metrics between different types of subnetworks. Especially, *LR* subnetworks tend to have a large scale, while *OS* subnetworks are associated with higher interconnectedness.

4.3 RQ6: How connected are the subnetworks?

Finally, we seek to explore the relationship between subnetworks.

Approach. To measure the inter-connection between two subnetworks, we define the *connection-strength*, i.e., the sum of the edge weights of the intersecting nodes of the two subnetworks. For example, the *connection-strength* between subnetwork A and subnetwork B is 10 (1+3+2+4), as shown in Figure 6.

$$Connection - strength = \sum weight_{intersection} \qquad (2)$$

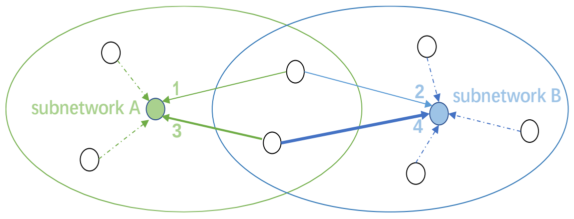

Figure 6: Example of computing the connection-strength.

We compute the *connection-strength* between each two of the 100 subnetworks. Then, we mark each connected subnetwork group using their categories as defined in RQ4. E.g., *OS* subnetwork and *LR* subnetwork should be marked as *OS-LR*. Note that we mark the subnetwork group as *Other* if one of the subnetworks belong to the *Other* category. To compare the differences of *connection-strength* between different types of subnetwork groups, we use the Kruskal-Wallis test [16], which is an extension of the Wilcoxon test and can be used to test the hypothesis that a number of unpaired samples originate from the same population.

Results. Based on the 100 subnetworks, we compute the *connection-strength* of 4,950 subnetwork groups, in which 11 subnetworks groups belong to the *Other*. When we look at the Top-100 subnetwork groups ranked by the *connection-strength*, we find that 30% groups belong to the *OS-LR*, 28% groups belong to the *OS-OS*, 14% groups belong to the *LR-LR* or *OS-TS*, and 3% groups belong to the *LR-TS* or *OS-AF*. Specifically, Table 4 shows the list of Top-20 subnetwork groups ranked by the *connection-strength*. We find that the *OS-OS* group ranks very high. Next, we compare the differences of *connection-strength* between different types of subnetwork groups, as shown in Figure 7. The distribution of *connection-strength* per type of subnetwork group is significantly different (Kruskal-Wallis test; p<0.001); Thus, we find that the types of subnetwork groups may have impact on the *connection-strength*.

Table 4: Top-20 subnetwork groups.

Subnetwork1	Subnetwork2	Group	Connection strength
ubuntu	alpine	OS-OS	7,487
ubuntu	debian	OS-OS	7,391
alpine	debian	OS-OS	5,665
ubuntu	centos	OS-OS	4,194
golang	alpine	OS-LR	2,833
ubuntu	basecontainer	Other	2,772
ubuntu	base	Other	2,550
centos	alpine	OS-OS	2,523
node	ubuntu	OS-LR	2,329
centos	debian	OS-OS	2,329
python	ubuntu	OS-LR	2,303
alpine	base	Other	2,095
ubuntu	java	OS-LR	1,925
alpine	basecontainer	Other	1,905
php	ubuntu	OS-LR	1,881
basecontainer	debian	Other	1,791
python	alpine	OS-LR	1,775
base	debian	Other	1,751
java	openjdk	LR-LR	1,688
ubuntu	openjdk	OS-LR	1,558

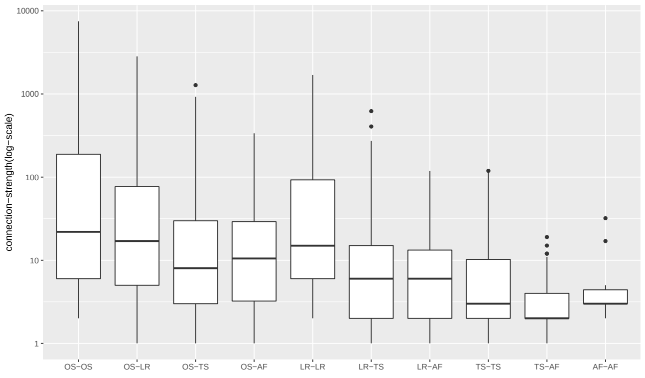

Figure 7: Comparison of the *connection-strength* between different subnetwork groups.

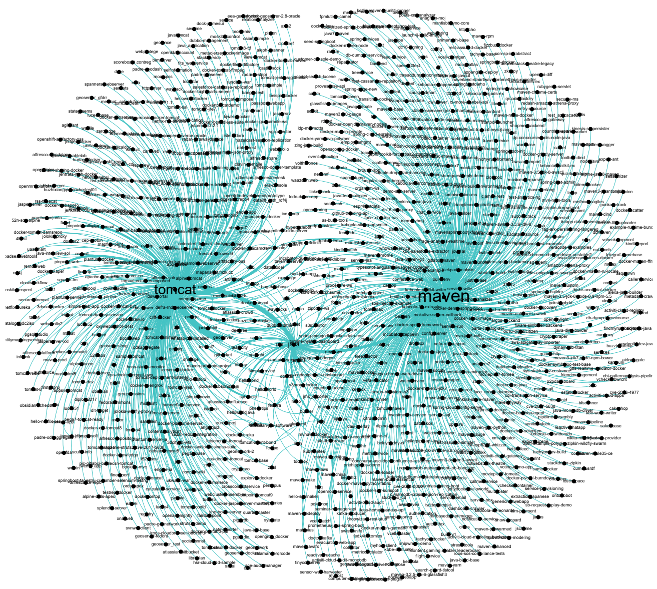

Figure 8: The joint network of Tomcat and Maven.

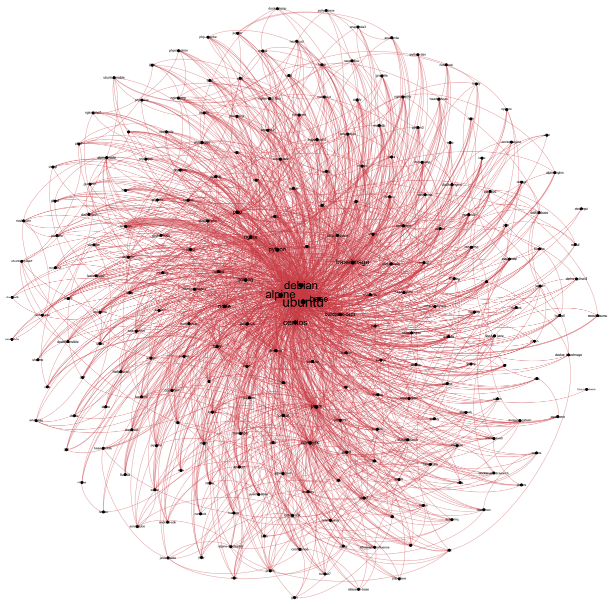

Figure 9: The joint network of Ubuntu and Alpine.

Those *OS-OS* subnetworks can be understood as docker containers which can be built on several different types of operating systems and run successfully due to the common unix kernel of these distributed operating systems. As for those *OS-LR* subnetworks, docker containers are built based on the basic functions support of system software, such as scheduling tasks, executing applications, and controlling peripherals and meanwhile the support of programming language designed to work on a variety of platforms. For those *LR-LR* subnetworks, docker containers are built because of different function modules may be more appropriate to use different languages. For those *OS-AF* and *OS-TS* subnetworks, operating systems manages all of the other application programs in a computer and platforms or application can't run without support of *OS*. Interestingly, *TS* or *AF* subnetworks nearly didn't interact with the same type of subnetworks as oneself, however, they are relatively more inclined to interact with subnetworks whose type are *OS* or *LR*. A potential explanation is that *TS* or *AF* are inclined to implement a specific function or developed to a specific field, however, *OS* and *LR* provide more foundational aids for software development and thus what they oriented is the whole software ecosystem. Meanwhile, subnetworks of *TS* or *AF* are not completely independent, they also interactive for cooperation targets in a relatively infrequent way.

After looking into the relationships between subnetworks, we find two common relationships: (1) *Cooperative relationship*: This pattern is particularly common in our dateset. Different types of containers solve different problems and domains so they always work together to accomplish complex functions. E.g., Tomcat cooperates with Maven to develop web pages, as shown in Figure 8. Also, some dockerfiles would have several FROM instructions at the same time, e.g., the front-end development uses PHP as base image, while after-end development uses java. (2) *Alternative relationship*: This pattern is more easily seen between the same type of subnetworks. Some containers are dependent on different subnetworks in this pattern because they can realize function through any of the subnetworks due to the similar functions. The relationship between similar container subnetworks, thus, is considered as an alternative relationship. Figure 9 gives the joint network of ubuntu and alpine, which both can provide basic systems for other containers but differs in size and community support. Significantly, reduce system size and runtime resource consumption accords with the lightweight technical feature of docker. The similarity between operating systems, additionally, can be applied to operating systems recommendations when building docker containers based on the principle of lightweight.

Answer to RQ6: There is a correlation between the connection strength of the two subnetworks and their specific category groups. Particularly, the subnetworks in the *OS-OS* group have higher inter-connection than other groups. Moreover, those linked subnetworks have cooperative or alternative relationship.

5 DISCUSSION

5.1 Implication

From our quantitative study results, we can distill some implications for different stakeholders.

For researchers. The ability to automatically identify technical dependencies, generate subnetworks and character structure, diversity and relationship between subnetworks of docker containers opens the door for many interesting avenues of research:

(1) *Container summary generation*: There are more than 6 million containers stored in the Docker Hub, having no tags or function descriptions is not conducive to understand or practical usage. Our

construction of docker dependency network provides the feasibility of acquiring knowledge through configuration files. Our instruction extraction and utilization of dockerfiles are conducive to automatic generation container summary.

(2) *Container recommendation*: We consider two kinds of recommendation applications for the docker containers. On the one hand, our empirical study finds alternative relationship between container subnetworks. It matters in size and build time when using alternative and lightweighter base image to build new container for software development, especially for those large scale build applications. Thus, deeper studies on alternative relationship are needed for recommending automatically alternative containers. On the other hand, reliable dependency relationship between container networks can be captured from existing dockerfiles and give some related container recommendations. A series of recommendations for related containers may quickly help developers to understand this container from these cooperated or atterative containers.

(3) *Container version technical debt*: In practice, a serious problem is the technical debt caused by the frequent changes of the container version. Our study reveals the function of the core container to improve software development and finds extremely popular OS containers that all identify their universal dependency relationship. Thus, impact on the whole docker ecosystem may be large if core container version changes. But how and even to what extent it influences to be further studied call the empirical on technical debt of docker container versions. Additionally, just-in-time recommendation of container version will also help to address this problem.

(4) *Container knowledge graph*: Our work mainly uses the base image to construct the container dependency networks. Researchers should integrate more dockerfile configuration data into this network, e.g., instruction complexity, container owner, project information, to form a container knowledge graph. Based on this knowledge graph, we can mine more knowledge and some patterns may become apparent.

For docker practitioners. Our tools, constructing automatically subnetworks and subnetwork pairs of docker containers and further provide a visual presentation with the aid of Gephi engine. This way can give docker practitioners the most intuitive feeling from network angle about a given container and even multiple containers. Visual and direct tools may be a huge attraction to incoming docker practitioners who can access quickly the associated other containers, including connection relationships, and to what extent with the provided containers and even deepen the understanding of this container through the associated containers.

For tool builders. Our study finds that very few dominant containers have an obviously greater influence on the overall docker ecosystem. Tool support for effective and accurate containers tags and summary even maintenance of dominant containers may aid stronger docker ecosystem appeal. We also find that the strongest subnetwork pairs type is *OS-OS* that indicates containers may have multiple alternative *OS* base containers. Tools recommending as small as possible basic *OS* containers under the premise of meeting the needs of developers will reduce build time in practice and in line with lightweight technical principles.

5.2 Threats To Validity

Threats to internal validity. The threats to internal validity relate to errors in our experimental data, tool implementation, and personal bias in container classification. To avoid errors during the experiment, we carefully test our data processing tool and verify the accuracy of our tool implementation by manually examining a large number of data instances outputted by each step of our tool. To reduce the personal bias in container classification, two authors do this classification independently and the tiny difference comparison results show the reliability of our manual classification.

Threats to external validity. One threat to validity is that our dataset consists of only 129,463 dockerfiles, the results may not be representative of all docker containers in Docker Hub. However, we make rigorous hypothesis tests on our analysis results. Besides, our approach capturing container pairs and constructing subnetworks hence can also be applied to other dockerfiles in Docker Hub. Another threat to validity is related to our selection of appropriate metrics to evaluate scale and interconnection in subnetworks, relationships between subnetworks. To mitigate this threat, we always use common and intuitive metrics and eliminate the unreasonable metrics through a large number of instances of visualization.

6 RELATED WORK

6.1 Docker and Dockerfiles

With the increase in relevance of Container concepts, such as docker, empirical research on the Docker-based software development process has received more attention. Zhang et al. [23] studied two prominent workflows, based on the automated builds feature on Docker Hub or Continuous Integration services, facing different trade-offs and this study further arises developers' focus on Docker-enabled workflows.

To promote the application of containers, more research focuses on the evolutionary, configuration, and build-quality of containers. Wu et al. [22] conducted a large-scale empirical study of build failures including frequency, fix effort, and evolution in the Docker context, and pointed that the failure rate and fix time of Docker build fluctuate and gradually increase across time. A clustering-based approach for mining Dockefile evolutionary trajectories proposed by Zhang et al. [24] revealed six distinct clusters of Dockerfile evolutionary trajectories. Hassan et al. [10] proposed RUDSEA, a novel approach to recommend updates of Dockerfiles to developers based on analyzing changes in software environment assumptions and their impacts.

While the existing literature helps researchers and practitioners gain a deeper understanding of Dockerfile usage and quality, few studies have investigated the docker container dependency, the only exception being the study by Cito et al. [5]. In that paper, the authors reported on an exploratory study with the goal of characterizing the Docker ecosystem, prevalent quality issues, and the evolution of Dockerfile. Their study was mainly based on sampling inspections of Top-100 and Top-1,000 most popular projects. Our work here is substantially different in two aspects. First, we consider a more refined set of studied dockerfiles, enabling powerful network construction and quantitative hypothesis testing, in addition to the basic statistics. Second, our goal is different, as we aim to comprehensively understand the characteristics, diversity, and relationship of the container dependency network.

6.2 Technical Dependency Network

With projects are developed and co-evolve with each other, research emphasis of code repositories has shifted from single project to complex software ecosystems [26]. The rise of technical dependency benefits from its ability to define ecosystem structure [13]. Much work has been done studying software evolution, ecosystems by static analysis of the project configuration files [9, 14, 19]. Ossher et al. [18] proposed analyzing import statements in Java source code to recover dependencies between the software projects of an ecosystem. Alexandre Decan et al. carries out a quantitative empirical on package dependency networks for seven packaging ecosystems of varying sizes and dependency networks tend to grow over time, both in size and in number of package updates. Zimmerman et al. [27] have applied incorporating metrics that are based on the dependency graph of a software system to improve prediction performance and Nguyen et al. [17] further studied the impact of dependency network measuring on software quality. They found that a small subset of dependency network measures have a large impact on post-release failure. Blincoe et al. [3] proposed reference coupling method to establish technical dependencies and identify ecosystems using it.

Despite the importance of docker in industry, to the best of our knowledge, no existing research has investigated the docker container from technical dependency network, nor empirically studied the structure, diversity, and relationship. With this paper, we attempt to address this research gap and provide insights into the docker dependency management.

7 CONCLUSION

In order to better understand the dependency network of docker containers, we construct the container dependency network from a set of more than 120,000 dockerfiles. We first investigate the characteristics of the container dependency network, including basic network metrics, degree distribution, and dominant containers. Next, for the Top-100 dominant containers, we construct their subnetworks and analyze the diversity and relationship among them. Our results help the researchers and practitioners gain a better understanding of the container dependency networks.

In future work, we plan to build visualization tools for the container dependency network. Also, we plan to investigate the evolutionary characteristics of the container dependency network during different time periods.

ACKNOWLEDGMENT

This work was supported in part by the National Key Research and Development Program of China (Grant No.2018YFB1004202), in part by the Foundation of PDL (Grant No.6142110190204), and in part by the Laboratory of Software Engineering for Complex Systems.

REFERENCES

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

[2] Chris Anderson. *The long tail: How endless choice is creating unlimited demand.* Random House, 2007.

[3] Kelly Blincoe, Francis Harrison, and Daniela Damian. Ecosystems in github and a method for ecosystem identification using reference coupling. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 202–211. IEEE, 2015.

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

[5] 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 *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 323–333. IEEE, 2017.

[6] Alexandre Decan, Tom Mens, and Maelick Claes. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, pages 1–4. ACM, 2016.

[7] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, 2019.

[8] Edmund A Gehan. A generalized wilcoxon test for comparing arbitrarily singly-censored samples. *Biometrika*, 52(1-2):203–224, 1965.

[9] Jesus M Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[10] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. Rudsea: recommending updates of dockerfiles via software environment analysis. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 796–801, 2018.

[11] Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. Mining best practices from dockerfiles: Towards better support for devops artifacts. In *Proceedings of the International Conference on Software Engineering (ICSE)*, page To be appeared. ACM, 2020.

[12] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[13] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 309–312, 2010.

[14] Mircea F Lungu. *Reverse engineering software ecosystems.* PhD thesis, Università della Svizzera italiana, 2009.

[15] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. Cliff's delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10(2):545–555, 2011.

[16] Patrick E McKight and Julius Najab. Kruskal-wallis test. *The corsini encyclopedia of psychology*, pages 1–1, 2010.

[17] Thanh HD Nguyen, Bram Adams, and Ahmed E Hassan. Studying the impact of dependency network measures on software quality. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10. IEEE, 2010.

[18] Joel Ossher, Sushil Bajracharya, and Cristina Lopes. Automated dependency resolution for open source software. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 130–140. IEEE, 2010.

[19] Francisco W Santana and Cláudia ML Werner. Towards the analysis of software projects dependencies: An exploratory visual study of software ecosystems. *IWSECO 2013 Proceedings*, pages 1–12, 2013.

[20] Ferdian Thung, Tegawende F Bissyande, David Lo, and Lingxiao Jiang. Network structure of social coding in github. In *2013 17th European conference on software maintenance and reengineering*, pages 323–326. IEEE, 2013.

[21] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 351–361. ACM, 2016.

[22] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. An empirical study of build failures in the docker context. pages 76–80, 2020.

[23] 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, 2018.

[24] Yang Zhang, Huaimin Wang, and Vladimir Filkov. A clustering-based approach for mining dockerfile evolutionary trajectories. *Science China Information Sciences*, 62(1):19101, 2019.

[25] Yang Zhang, Huaimin Wang, Gang Yin, Tao Wang, and Yue Yu. Social media in github: the role of @-mention in assisting software development. *Science China Information Sciences*, 60(3):032102, 2017.

[26] Yang Zhang, Yue Yu, Huaimin Wang, Bogdan Vasilescu, and Vladimir Filkov. Within-ecosystem issue linking: a large-scale study of rails. In *Proceedings of the 7th International Workshop on Software Mining*, pages 12–19, 2018.

[27] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540, 2008.