Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 4592428, 14 pages
https://doi.org/10.1155/2022/4592428

Research Article

Burner: Recipe Automatic Generation for HPC Container Based on Domain Knowledge Graph

Shuaihao Zhong[1], **Duoqiang Wang**[1], **Wei Li**[2], **Feng Lu**[1] **and Hai Jin**[1]

[1]*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China*
[2]*Australia-China Joint Research Centre for Energy Informatics and Demand Response Technologies, Centre for Distributed and High Performance Computing, School of Computer Science, University of Sydney, Sydney, Australia*

Correspondence should be addressed to Shuaihao Zhong; shuaihaozhong@hust.edu.cn

Received 25 February 2022; Revised 4 April 2022; Accepted 20 April 2022; Published 25 May 2022

Academic Editor: Yingjie Wang

As one of the emerging cloud computing technologies, containers are widely used in academia and industry. The cloud computing built by the container in the high performance computing (HPC) center can provide high-quality services to users at the edge. Singularity Definition File and Dockerfile (we refer to such files as recipes) have attracted wide attention due to their encapsulation of the application running environment in a container. However, creating a recipe requires extensive domain knowledge, which is error-prone and time-consuming. Accordingly, more than 34% of Dockerfiles in Github cannot successfully build container images. The crucial points about recipe creation include selecting the entities (base images and packages) and determining their relationships (correct installation order for transitive dependencies). Since the relationships between entities can be expressed accurately and efficiently by the knowledge graph, we introduce knowledge graph to generate high-quality recipes automatically. This paper proposes an automatic recipe generation system named Burner, enabling users with no professional computer background to generate the recipes. We first develop a toolset including a recipe parser and an entity-relationship miner. Our two-phase recipe parsing method can perform abstract syntax tree (AST) parsing more deeply on the recipe file to achieve entity extraction; the parsing success rate (PSR) of the two-phase parsing method is 10.1% higher than the one-phase parsing. Then, we build a knowledge base containing 2,832 entities and 62,614 entity relationships, meeting the needs of typical HPC applications. In the test of image build, the singularity image build success rate reaches 80%. Compared with the ItemCF recommendation method, our recommendation method TB-TFIDF achieves a performance improvement by up to 50.86%.

1. Introduction

With the rapid development of IoT technology, the application scenarios are very wide. When the computing power of edge computing is limited in IoT applications, high-performance computing cloud can supplement the powerful computing power. Container technology is widely popular due to its lightweight and convenience. At the same time, researchers in HPC have also recognized the value of containers. Singularity [1] is currently the most widely used container technology in the HPC field, and many optimizations have been made for HPC applications. First, Singularity can prevent user privilege escalation within the container. Secondly, it can make full use of the host's high-speed interconnect hardware such as InfiniBand, simplifying access to acceleration devices such as GPUs. By now, most of the world's top HPC centers use Singularity as a solution for containerizing HPC applications in production environments.

Container technology simplifies the packaging of applications so that the dependent environment can be easily maintained. The encapsulation of the application running environment in container technology depends on recipes.

The recipe is the core of implementing application-dependent environment encapsulation which is a script written based on domain-specific language (DSL) for building container images. It records all instructions on how to build the application running environment. The use of recipes improves the transparency of the research process and facilitates the reproduction of scientific research results [2–4]. However, the effort involved in manually constructing an environment specification is non-trivial. An experienced developer may spend 20 minutes to 2 hours creating a recipe for an application and often fails to build an accurate specification [5]. Common challenges in writing recipes include selecting base images and packages and determining the correct installation order for transitive dependencies.

Henkel et al. designed the Binnacle toolset [6] to parse the 178,000 Dockerfiles present in the collected Github projects. This toolset is capable of mining semantic rules and best practices in Dockerfiles, providing friendly suggestions to Dockerfile developers. Unfortunately, the Binnacle toolset cannot be directly applied to Singularity recipe parsing, nor can it mine dependencies between packages. DockerizeMe [7] reproduces the running environment of Python code by building a Docker image and uses a combination of static analysis and dynamic analysis to solve the import error problem in Python. However, DockerizeMe mainly analyzes the Python language, which is only suitable for specific scenarios and cannot deal with the diversity of software systems.

HPC Container Maker [8] is an open source tool to make it easier to generate container specification files. HPCCM can generate Dockerfiles or Singularity Definition Files from a high level Python recipe. However, HPCCM essentially uses the Python to define a set of its own recipe specifications, which has relatively high requirements for users. On the other hand, due to the lack of domain knowledge, HPCCM cannot provide users with recommendations for key entities. Therefore, users must have relatively professional computer knowledge (such as Python and recipe syntax specifications) to implement customized recipes for HPC applications.

There are two challenges to realize the automatic generation of recipes in the HPC field:

(i) (C1) How to parse recipe files and extract entities and entity relationships from them

(ii) (C2) How to apply the obtained entities and entity relationships to the automatic generation of recipes

To address (C1), we first design and implement a two-phase parsing method for Singularity recipes and a relationship miner to extract key entities of recipes and mine relationships between entities. For (C2), we consider that the dependencies of software packages can be expressed more efficiently with graph data structures, so we store the acquired knowledge in a standardized graph database such as Neo4j. The knowledge graph provides data support for the automatic generation of recipes, which has anexcellent scalability. In the automatic generation of recipes, we improve the tag-based recommendation method to meet HPC users' personalized and diverse needs.

In summary, we make four core contributions:

(1) A unique toolset is designed for Singularity recipes to automatically extract the knowledge required for image construction and mine the associations between entities

(2) We build a knowledge graph of HPC containers to provide support for automatic recipe generation. The knowledge graph also provides functions such as entity recognition and entity-relationship query

(3) An improved recommendation method based on TF-IDF is designed, significantly improving recommendation performance

(4) Burner: an automatic recipe generation system. It is worth mentioning that Burner supports both Singularity Definition File and Dockerfile rule specifications

The original recipe dataset and parsing results can be obtained at https://github.com/jhshz520/BurnerRecipe. The rest of this paper is organized as follows: Section 2 reviews related work. Section 3 presents the overall design of the Burner. Section 4 introduces the construction of domain knowledge graph, mainly including two-phase parsing of recipes, entity extraction and entity relationship mining. The automatic generation of recipes based on knowledge graphs is described in Section 5. Section 6 is the performance evaluation of our toolset and Burner system. The last section draws conclusions and proposes future work.

2. Related Work

2.1. Container Technology in HPC. Charliecloud is an open source software based on the user-defined software stack (UDSS), emphasizing that it can be executed without users having root permissions. Charliecloud is a lightweight container implementation with a small code size of only about 800 lines. However, its functions, portability, and dependencies are slightly insufficient, and it cannot provide a powerful reproduction mechanism [9]. NERSC cooperated with Cray to develop Shifter [10, 11]. The main idea of Shifter is to reuse some components of Docker workflow and improve the runtime engine to meet the needs of HPC applications. Shifter reuses key components of the Docker ecosystem, rewriting the Docker runtime. However, the setup and management of Shifter are also relatively complex. Sarus [12] builds around the OCI specification, uses runc as the container runtime, and extends the functionality of HPC use cases by using OCI Hook, but it is not much different from Charliecloud and Shifter, all of them need to be used with the modified Docker containers to achieve targets for applications in HPC. Singularity is currently the best container solution in the HPC environment. It has a unique security model that allows untrusted users to safely run untrusted containers on multi-tenant systems. A special image format Singularity Image Format (SIF) is used to package and distribute containers. This compressed single-layer image format greatly reduces the storage space of the image and

facilitates the distribution of the image with better performance. In addition, Singularity implements cryptographic signature and verification with excellent portability and repeatability [13]. Singularity allows user-defined/managed/created containers to be easily integrated into existing HPC workflows and also provides compatibility with older OS versions via the setuid launcher. As of December 2021, Singularity has three major version iterations with many useful features.

2.2. Recipe Analysis. Singularity appeared in recent few years, and the application scenarios are not as extensive as Docker. At present, there is still a lack of research on Singularity Definition File, but the existing researches on Dockerfile have great reference significance. Cito et al. [14] conducted an exploratory analysis of the Docker container ecosystem on Github, and the research dataset contained more than 70,000 Dockerfiles. After comparing the most popular top100 and top1000 projects, it was found that up to 34% of the Dockerfiles in these projects could not be successfully built due to various problems and 28.6% of the quality problems were caused by the lack of version tags. Schermann and Zumberi [15] collected structured data about the status and changes of Dockerfiles from over 15,000 projects on Github and stored them in a PostgreSQL database. Zhang et al. [16] studied the impact of Dockerfile evolution trajectory on Dockerfile quality and corresponding image build latency. It was found that the fewer the number of image layers and the larger the space occupied by each layer of images, the fewer image quality problems and the shorter the build latency. By using the Dockerfile Lint tool Hadolint [17] to perform static analysis on a large number of Dockerfiles, Lu et al. discovered a problem in Dockerfiles that they called "Temporary File Smell." In the process of image building using Dockerfile, due to Docker's Copy On Write mechanism, inappropriate writing order of Dockerfile instructions will result in redundant temporary files in the Docker image [18, 19]. Yin et al. [20] proposed the STAR method, which solved the tag recommendation problem for Docker image repositories without training data. Hassan et al. [21] developed the Rudsea tool, which could implement Dockerfile update prompts based on analysis of changes in the software environment.

2.3. Software Domain Knowledge Graph. Knowledge graphs are not only widely used in search engines, question answering systems [22], and medical service support, but also play an important role in software reuse. Lin et al. formed an intelligent development environment IntelliDE [23] by aggregating, mining and analyzing software big data, and providing assistance to developers in the software development life cycle. DoekerPedia [24] proposed by Osorio et al. is the first known knowledge graph related to Docker images. Clair [25] is used to detect vulnerabilities in image instances to obtain information about software package versions and their vulnerabilities. DockerizeMe proposed by Horton et al. [7] builds a knowledge base of dependencies between Python packages and APT packages.

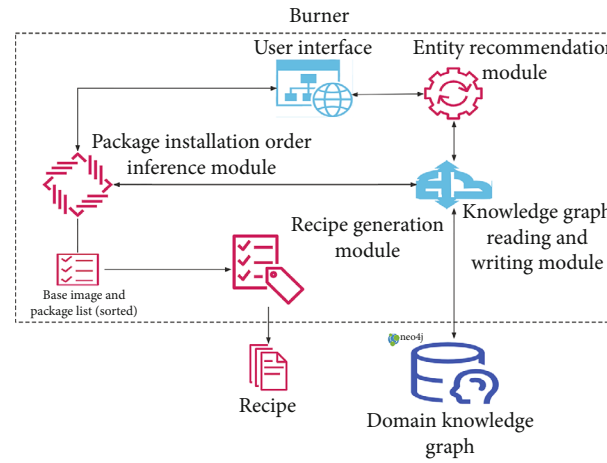


Figure 1: Design of Burner.

3. Burner

The main purpose of Burner is to solve the problem of automatic recipe generation. We solve the dependency problem by building an offline knowledge base and design an inference algorithm to return dependencies in a feasible installation order. We use the Django framework to implement Burner as a web application that researchers can use by visiting the website.

As shown in Figure 1, Burner uses the knowledge graph to automatically generate recipes. The core modules of Burner include the knowledge graph reading and writing module, which mainly provide data support for other modules. The entity recommendation module can recommend entities such as base images and software packages according to the Tag selected by a user. The software package installation order inference module can infer the order of the software package entities selected by the user to form an ordered software package installation list. Finally, the recipe generation module generates instructions according to the rules of the recipe, and saves the generated instructions in the form of files.

4. Construction of Knowledge Graph in HPC Container Domain

The knowledge graph is the cornerstone for our automatic generation of recipes and can provide strong support for dependency inference. Therefore, we start with the construction of domain knowledge graph to illustrate our work. As shown in Figure 2, the construction of knowledge graph in the field of HPC container mainly includes four parts: raw data acquisition, recipe parsing, knowledge fusion, and knowledge storage.

4.1. Ontology of Knowledge Graph. The ontology of the knowledge graph in the HPC container domain is shown in Figure 3, which includes 4 entity types and their attributes and 8 entity relationships.

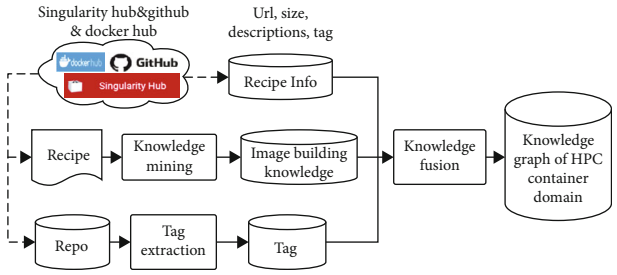


Figure 2: Overview of knowledge graph construction.

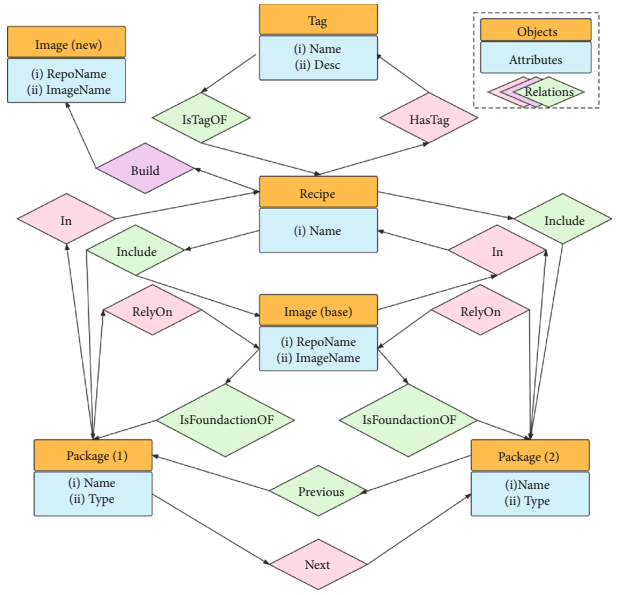


Figure 3: Ontology of knowledge graph in HPC container domain.

4.2. Data Acquisition. The data mainly comes from Singularity Hub, a publicly available platform for building and deploying scientific containers, which provides great convenience for reproducible scientific fields [26]. We use a customized crawler to collect and organize the recipes and their authors, tags, and other information. The raw data we obtained contains 530 tags and more than 1000 published recipes for HPC applications.

4.3. Two-Phase Parsing of Recipe. Code 1 is an example of a Singularity Definition File, where bash statements are usually nested [27]. We design a two-phase parsing method to parse the nested bash statements in recipe. The first stage is the instruction parsing, and AST parsing is performed according to the grammar specification defined by Singularity. The second stage parses the bash statements nested in the ASTs.

The first stage identifies each instruction according to the grammar of the recipe. The Singularity Definition File has different instruction blocks. Except for *Bootstrap* instruction and *From* instruction, all other instructions start with %. Therefore, regular expressions can be used to match and divide instructions, and each instruction can be parsed into an AST node, as shown in Figure 4.

```
1 Bootstrap: docker
2 From: ubuntu:16.04
3
4 %post
5 apt-get -y update
6 apt-get -y install fortune cosway lolcat
7
8 %runscript
9 fortune | cowsay | lolcat
```

Code 1: Singularity Definition File fragment named lolcow.def..

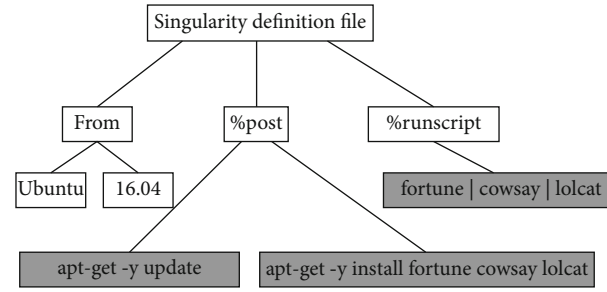


Figure 4: The first stage of AST parsing.

Table 1: 50 most commonly used Bash commands in recipes.

Categories	Name
System management	adduser, useradd, sudo, groupadd, nproc
File management	cp, rm, chmod, find, ln, chown, mv, mktemp
Disk management	cd, mkdir, pwd
System setting	set, export, gpg, ldconfig, sha256sum
Backup and compression	tar, unzip
Document editing	grep, sed, echo, wc
Package management tool	apt, apt-get, apt-key, apt-add-repository, yum, npm, yarn, gem, dpkg, dnf
Download tool	wget, curl, git, pip
Script run tool	bash, sh, python, php, go
Build tool	make, cmake, config
Reserved word	true

In the second stage of parsing, through command analysis, it is known that the information of the base image is in the "From" command field of the recipe and information of the packages is mainly in the "%post," "%environment," and "%runscript" instructions. However, the bash statements are often nested in the "%post" and "%runscript," which are numerous and varied.

It is impractical to design corresponding parsing methods for all Bash instructions, so we classified and counted these Bash instructions and found that 80% of the Bash command line calls are included in the 50 most commonly used commands. The names and classifications of the 50 commands are shown in Table 1. In this paper, we design a Bash statement parser for these 50 instructions by

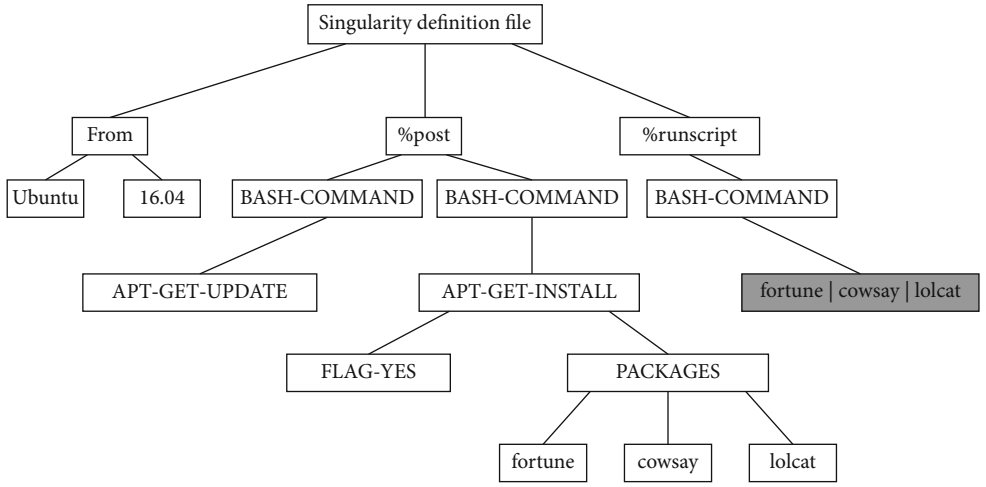


Figure 5: The second stage of AST parsing.

Input: packages_list, min_support, minConf
Output: package_pair_list
1 C1 = create_C1(packages_list)
2 L1, support_data = generate_L1_by_C1(packages_list, C1, min_support)
3 L = [L1]
4 C2 = aprioriGen(L[0])
5 L2, support_data = scanPackagelist(packages_list, C2, min_support)
6 L = L.append(L2)
7 package_pair_list = generateRules(L, support_data, minConf)

Algorithm 1: Software Package Association Mining Algorithm.

referring to the command manual and official documents of these instructions. The Bash statement parser is implemented by modifying the shellcheck tool [28].

As shown in Figure 5, after the second stage of parsing, the AST is generated. The parsing of commonly used Bash statements greatly enriches the content of the abstract syntax tree, which also provides a foundation for the extraction of software package entities and the mining of dependencies.

In the parsing example, *apt* – get – *y update*, *apt* – get – *y installfortunecowsaylolcat*, and *fortune*|*cowsay*|*lolcat* cannot be parsed in the first stage, but in the second stage, *apt* – get is one of the common commands, which can be further recognized and parsed by the Bash statement parser, while *fortune*|*cowsay*|*lolcat* cannot be further parsed because it is not a common command.

4.4. Entity Extraction and Entity Relationship Mining. Entity extraction can be performed from the ASTs generated by parsing. The entities we focus on are mainly base images and software packages. The information about a base image can be obtained from the child nodes of *From* node. We can traverse the subtree with %*post* and %*runscript* nodes as the root node to find the PACKAGES nodes for package information. During the traversal process, we can obtain the installation method of the packages from nodes such as APT-GET-INSTALL, YUM-INSTALL, and PIP-INSTALL. While traversing the tree, the appearance order of software packages in the PACKAGES node is also recorded, which is convenient for subsequent mining of software package dependencies.

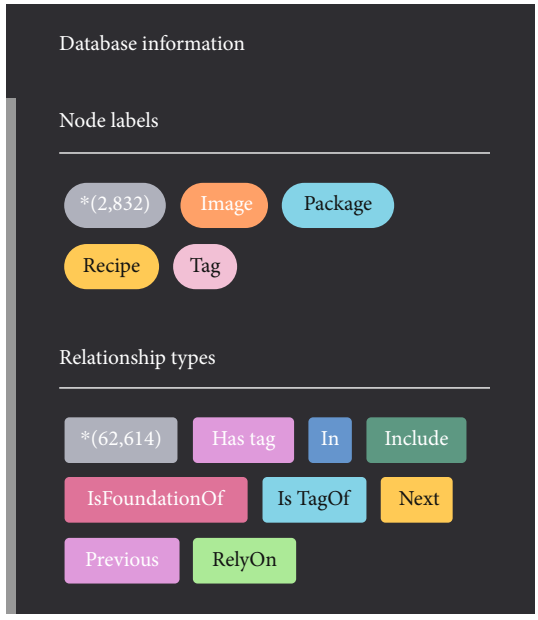


Figure 6: Information of Neo4j domain knowledge graph.

Table 2: Entities and their properties.

Node type	Attributes	Explanation
Recipe	Name	The name of the recipe
Image	RepoName	The name of the project where the base image is located
Image	ImageName	The name of the base image
Package	Name	The name of the package
Package	Type	The type of the package installation method, there are four types in total: pip, pip3, apt-get, and yum
Tag	Name	The name of the tag
Tag	Desc	The description of the tag

Table 3: Entity relationships.

Relation type	Explanation
Include	The relationship between recipe and its base image or software package, indicating that recipe contains a certain base image or a software package
In	The relationship between base image or software package and recipe, indicating that the base image or software package is included in the recipe
RelyOn	The relationship between package and base image means that the package depends on the base image for installation and configuration
IsFoundationOf	The relationship between base image and package, which means that the image is a starting image for the package to install and configure
IsTagOf	The relationship between tag and recipe, indicating that the tag is a label of the recipe
HasTag	The relationship between recipe and tag, indicating that the recipe has the tag it points to
Previous	The relationship between packages, indicating that the current package appears in the recipe before the package it points to
Next	The relationship between packages, indicating that the current package appears in the recipe after the package it points to

The main purpose of the software package entity association mining is to find the predecessor and successor relationships between software packages [29]. We use the Apriori algorithm to mine package dependencies [30]. If the confidence level of the association rule $pkg1 \longrightarrow pkg2$ is 1.0, it means that $pkg2$ can be installed under the condition that the package $pkg1$ is known to be installed; then, we can consider that the package $pkg2$ is a dependent package of the package $pkg1$. The mining algorithm is shown in Algorithm 1. We set min_support as the reciprocal of the minimum frequency of software packages so that the dependencies between software packages can be mined to the greatest extent. The minimum confidence is set to the most commonly used 0.8.

4.5. Knowledge Fusion and Knowledge Graph Construction. Knowledge fusion [31] is to unify and standardize the knowledge extracted from different recipes. The acquired knowledge is uniformly encoded with all entities as nodes and all entity relationships as edges. This unified code is the unique identification of the entity or entity relationship in the knowledge graph. Finally, the knowledge is stored in the graph database Neo4j (see Figure 6), which contains 2832 entities and 62614 relationships. The standardized knowledge base can provide support for the customized generation of subsequent recipes. The entities and their attributes are shown in Table 2, and the entity relationships are shown in Table 3.

5. Implementation of Burner

The most important modules of Burner are the entity recommendation module and the installation order inference module. In the recommendation module, we improve the tag-based recommendation method, which can avoid the influence of popular tags and popular items on the recommendation effect. In the installation order inference module, we use a graph algorithm to supplement package dependencies and determine the package installation order.

5.1. Tag-Based Base Image and Package Recommendation. Tag is the label that the user marks on the recipe, but a recipe contains a base image and multiple package entities. There is no direct relationship between tags and these entities. The most commonly used software such as *git* and *wget* are widely present in recipes. How to find the entity in the recipes that can well represent the Tag is a problem worth thinking about.

In Figure 7, we count the number of occurrences of tags; it can be observed that tags conforms to the long-tailed distribution [32]. In order to better meet the needs of user personalization, we design a tag-based recommender system inspired by the idea of TF-IDF [33]. The simplest tag-based recommendation method counts the number of times tagged by tags to recommend items. In the actual system, according to the tags selected by the user, the corresponding most popular items are searched for the recommendation. However,

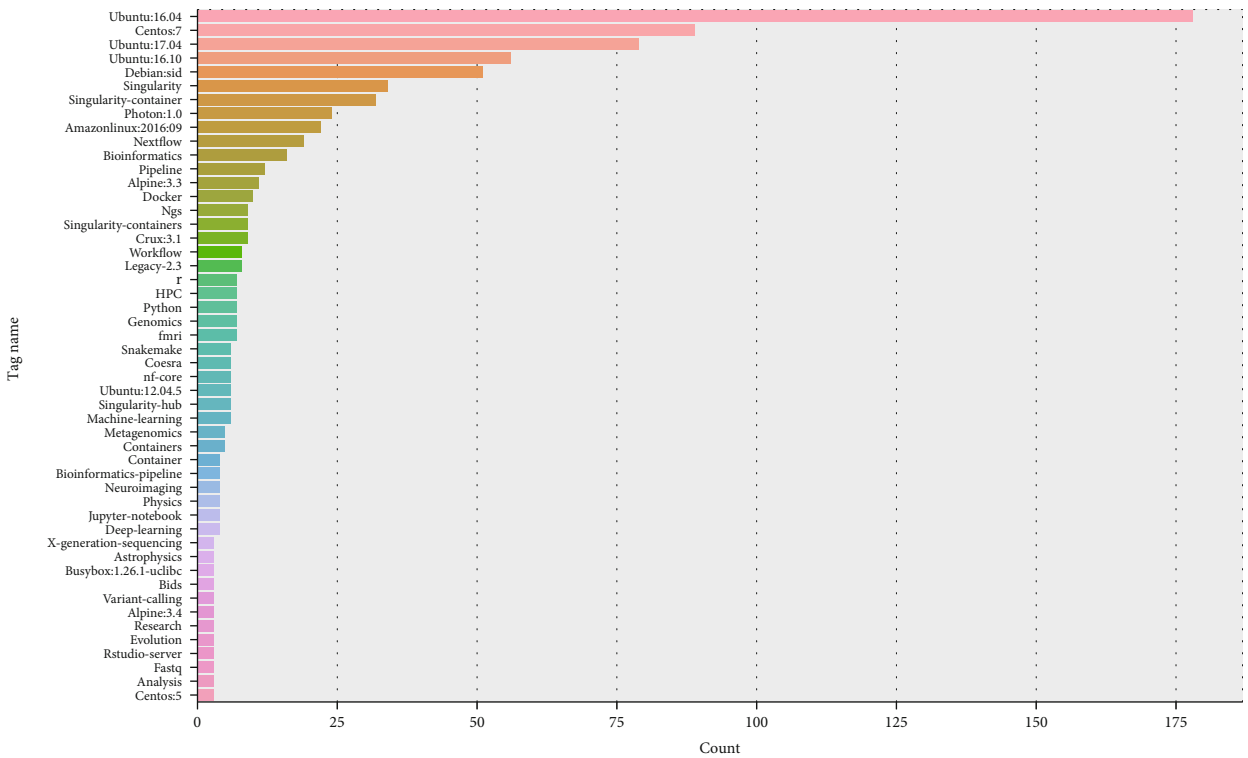


Figure 7: Statistics of Tag occurrences.

the apparent disadvantage of this recommendation method is that popular tags and popular items have a considerable weight, which dramatically reduces the novelty of the recommendation results. To this end, we have optimized the tag-based recommendation algorithm by drawing on the idea of TF-IDF. The core of our recommendation algorithm is based on the fact that a software package has appeared under a certain tag and hardly appears in other tags, so it can be considered that the software package is the core package under this tag.

As shown in Formulas (1), (2), and (3), $n_{i,j}$ represents the number of times that pkg_i is marked with tag_j, and $\sum_k n_{k,j}$ represents the total number of times that all software packages are marked with tag_j. $|Tag|$ indicates the total number of tags, $|\{j : pkg_i \in t_j\}|$ represents the number of tags used to mark pkg_i by the user. In Formula (3), $|\{j : pkg_i \in t_j\}| + 1$ can prevent the denominator from being 0:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}, \tag{1}$$

$$idf_i = \lg \frac{|Tag|}{|\{j : pkg_i \in t_j\}| + 1}, \tag{2}$$

$$Weight_{i,j} = tf_{i,j} \times idf_i. \tag{3}$$

5.2. Dependency Complement and Order Inference. Algorithms 2 and 3 show the complete process of package dependency complementation and package order inference. Algorithm 2 is implemented based on depth-first search

```
Input: core_pkgs
Output: complete_pkgs_set
1 foreach pkg in core_pkgs do
2     dp_pkgs = searchPredecessorByDFS(pkg)
3     complete_pkgs_set.add(dp_pkgs)
4 end
```

Algorithm 2: Package Dependency complementary Algorithm.

(DFS) by taking advantage of the transitive nature of package dependencies. After the user specifies the core packages related to the application, some dependency packages that these core packages depend on may not be included. The representation of package dependencies in the graph is that there is a directed edge between the package and the dependent package. After obtaining the dependencies of the software package list to be installed, the inference module will add dependency packages together with the core software packages specified by the user to the final set of software packages to be installed.

Figure 8 shows the possible dependencies of software packages in the graph. In actual use, we only consider the relationship of Previous, because Previous and Next appear in pairs and their functions are equivalent. The packages pointed to by the edges of Previous in the subgraph are in the first order.

The inference of the software package order in Algorithm 3 is to use the topological sorting method to sort all the software packages to be installed. The core idea of

```
Input: complete_pkgs_set
Output: sorted_pkgs_list
sub_graph = exractSubGraphFromKG(complete_pkgs_set)
out_degree_count = countOutDegreeForGraph(sub_graph)
while sub_graph is not empty do
    foreach pkg in zeroOutDegree(sub_graph) do
        sorted_pkgs_list.append(pkg)
        foreach pkg_next in nextNode(pkg) do
            pkg_next_out_degree -=1
        end
        remove_node(pkg, sub_graph)
    end
end
```

Algorithm 3: Package installation order inference Algorithm.

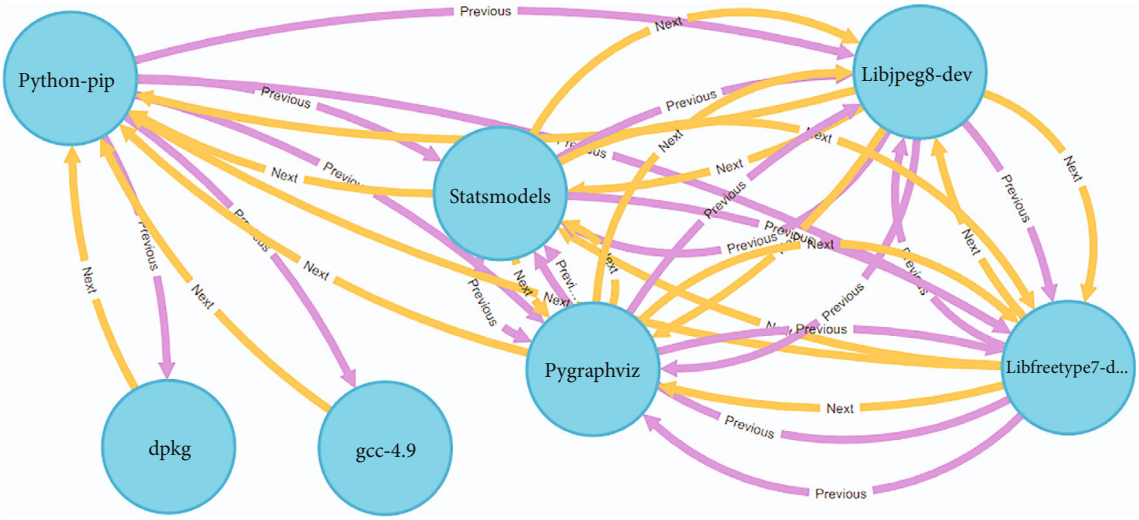


Figure 8: Example of a subgraph of package dependencies in Burner.

topological sorting is to continuously remove the nodes with zero out-degree in the subgraph of the software package until the subgraph is empty. If a node in the subgraph has no out-degree, the software package represented by this node can be installed directly without depending on other software packages. During the iteration, the node with out-degree zero is added to the sorted list, and all dependent edges of the node are removed from the subgraph. By repeating the above process until the graph is empty, the installation order of the packages is finally obtained.

6. Evaluation

6.1. Burner Demonstration. Burner is very friendly for HPC users even with no computer background. In the process of generating recipes using Burner, users do not need to perform any text input operations; during the entire interaction, they can complete the generation of customized recipes only by selecting operations.

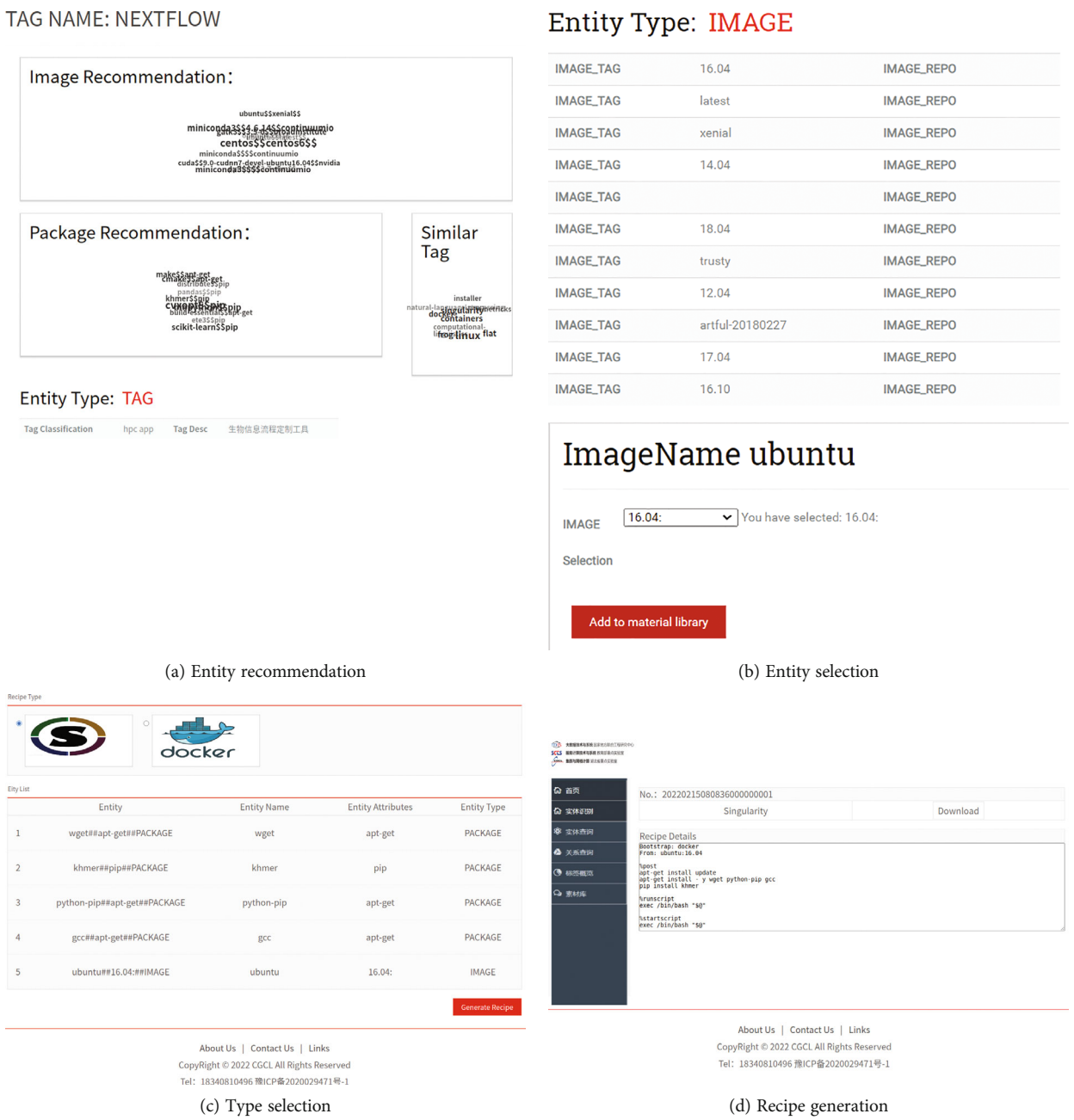
In Figure 9, we take the *nextflow* tag as an example to demonstrate the generation of Singularity Recipe. Figure 9(a) shows the recommendation of software packages and base images based on the tags selected by the user, and the recommended entities are displayed in a dynamic word cloud. Then, the users can select the required base image and software packages to add them to the material library (see Figure 9(b)), and the back end of the material library use the Redis database to quickly perform operations such as additions and deletions. As shown in Figure 9(c), after completing the selection of materials, the users can also specify the type of recipe. Currently, the system supports Singularity Definition File and Dockerfile. Users can preview the generated recipe online or perform operations such as edit, download, and delete (see Figure 9(d)).

6.2. Metrics

6.2.1. Evaluation Metrics for Recipe Parser

$$PSR = \frac{|Node_{total}| - |Node_{unknown}|}{|Node_{total}|}. \quad (4)$$

To quantify the performance of our recipe parser, we define a parsing success rate in this paper. After the



(a) Entity recommendation

(b) Entity selection

(c) Type selection

(d) Recipe generation

Figure 9: Recipe generation demonstrate.

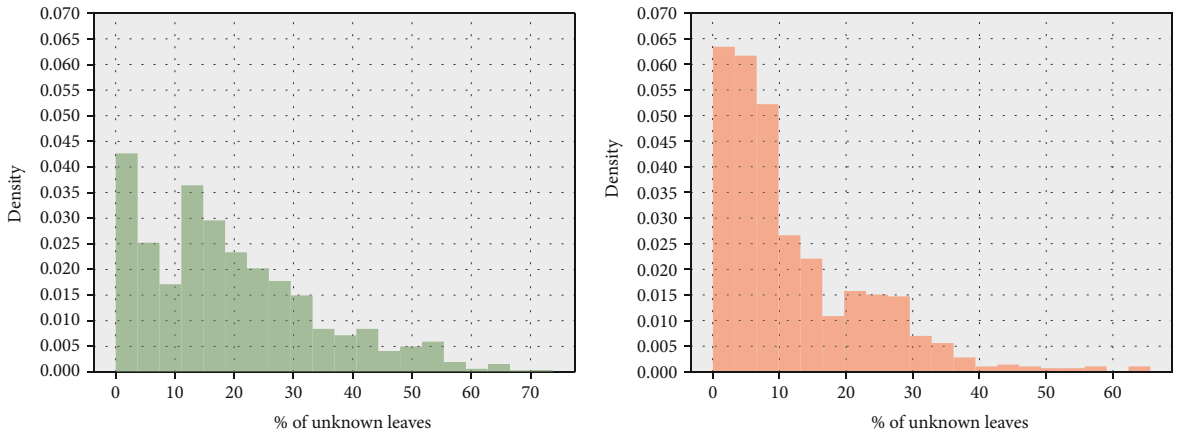
parsing is completed, we uniformly mark the nodes that cannot be parsed as UNKNOWN (gray nodes in Figures 4 and 5). We use $|Node_{total}|$ represents the total number of nodes in the AST, and $|Node_{unknown}|$ represents the number of UNKNOWN nodes. The definition of parsing success rate (PSR) is shown in Formula (4), a larger value of PSR means that the parser has stronger performance.

6.2.2. Evaluation Metrics for Recommendation Performance. In the practical application, we use tags for *TopN* recommendation [34]. Recommended entities include base images and packages. We use the mainstream *Precision*, *Recall* and *F1* to measure the recommendation performance of the system. The definitions of recommendation metrics are shown in Formulas (5), (6), and (7):

$$Precision = \frac{\sum_{u \in U} |R(u) \cap T(u)|}{\sum_{u \in U} |R(u)|}, \tag{5}$$

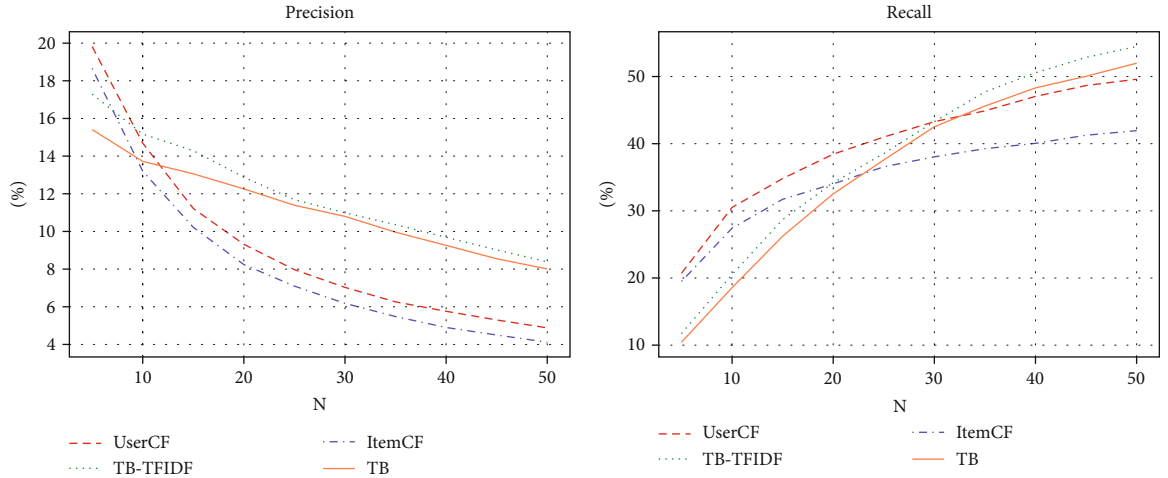
$$Recall = \frac{\sum_{u \in U} |R(u) \cap T(u)|}{\sum_{u \in U} |T(u)|}, \tag{6}$$

$$F1 = \frac{2 \times precision \times recall}{precision + recall}. \tag{7}$$



(a) Stage I (b) Stage II

Figure 10: Density histograms of UNKNOWN nodes for two parsing stages.



(a) Precision (b) Recall

Figure 11: Precision and recall of four recommendation methods.

6.2.3. Evaluation Metrics for Image Build

$$BSR = \frac{|Recipe_{bs}|}{|Recipe_{total}|}. \tag{8}$$

We use the build success rate (BSR) as a functional indicator of the system. Whether the image can be successfully built can intuitively represent the quality of the generated recipes. The definition of BSR is shown in Formula (8). $|Recipe_{total}|$ represents the total number of generated recipes, and $|Recipe_{bs}|$ represents the number of recipes that can successfully build the container images.

6.3. Experimental Results and Analysis

6.3.1. Two-Phase Parsing Method. We performed statistical analysis on ASTs generated by 1000 recipes. The density histograms of the distribution of UNKNOWN nodes in two parsing stages are shown in Figure 10.

After the first stage of parsing, 28.3% of the nodes are marked as UNKNOWN as shown in Figure 10(a). As shown

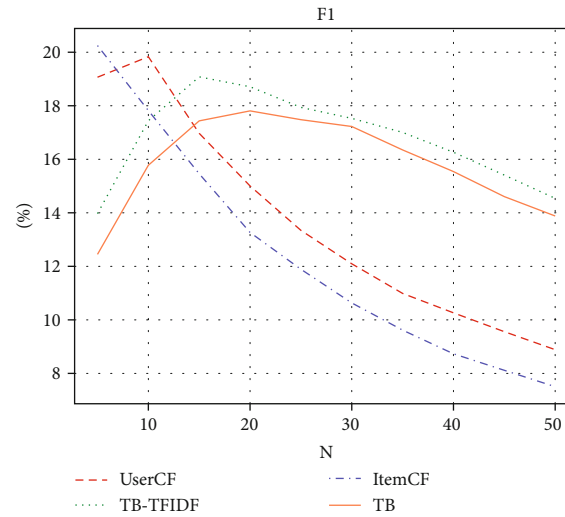


Figure 12: F1 of the four recommendation methods.

Table 4: Performance comparison of the four recommendation methods.

N	Precision				Recall				F1			
	UserCF	ItemCF	TB-TFIDF	TB	UserCF	ItemCF	TB-TFIDF	TB	UserCF	ItemCF	TB-TFIDF	TB
5	19.82	18.66	17.29	15.41	20.7	19.49	11.72	10.44	19.066	20.25	13.97	12.447
10	14.69	13.2	15.17	13.72	30.51	27.42	20.51	18.56	19.832	17.821	17.44	15.777
15	11.22	10.22	14.29	13.06	34.81	31.72	28.7	26.22	16.97	15.459	19.08	17.435
20	9.32	8.24	12.88	12.26	38.44	34.01	34.18	32.53	15.003	13.266	18.71	17.808
25	7.97	7.1	11.69	11.39	40.99	36.56	38.54	37.57	13.345	11.891	17.939	17.48
30	7.03	6.18	10.99	10.8	43.28	38.04	43.28	42.52	12.095	10.633	17.529	17.225
35	6.26	5.48	10.36	9.96	44.89	39.25	47.71	45.6	10.988	9.617	17.004	16.349
40	5.76	4.9	9.69	9.26	47.04	40.05	50.56	48.31	10.263	8.732	16.263	15.541
45	5.3	4.5	9.02	8.55	48.66	41.26	52.85	50.04	9.559	8.115	15.409	14.605
50	4.88	4.12	8.39	8.01	49.6	41.94	54.47	51.99	8.886	7.503	14.54	13.881

in Figure 10(b), in the second stage, the density of recipes with high PSR increases significantly, and the PSR in some ASTs even reaches 100%. On average, only 18.2% of the nodes could not be parsed in the second stage, and the PSR increased by 10.1%. Then, we analyzed the recipes with low PSR and found that the main reason was that some Bash commands in these recipes were not commonly used or the Bash statements were nested too deeply. The results show that our two-phase parsing method can effectively perform recipe parsing and entity extraction.

6.3.2. Recommendation Performance Test. Taking recommending software packages to users as an example, we compare the performance of four recommendation methods. The four methods are UserCF, ItemCF, TB, and TB-TFIDF. UserCF and ItemCF do not use tag information, only use user and item information as input. TB simply uses statistical information, and TB-TFIDF uses TF-IDF to greatly improve TB. There are two hyperparameters K and N in UserCF and ItemCF. K represents the selection of K users with the most similar interests to the recommended user, and N represents the number of items recommended to the user. After grid search and tuning, the optimal value of K is set to 80. Figures 11 and 12 show the performance of the four recommendation methods under different N values; the detailed results are shown in Table 4.

It can be observed that with the increase of N, the *Precision* of UserCF and ItemCF has a relatively significant decline. The recommendation method of TB-TFIDF is relatively stable, and the *Precision* is usually above 10%. From the $F1$ value that measures the overall performance of the recommender system, the effect of TB-TFIDF is also the best. The average number of packages contained in each recipe is 23. In practical applications, we set the N value to 20 or 25. Experiments show that the recommendation performance TB-TFIDF is best. The TB-TFIDF recommendation method also does not have the problem of cold start, which is more in line with the actual application scenario.

6.3.3. Image Build Test. In the image build test, 50 different tags are selected from the tag list by executing a random function. For each Tag, entity recommendation and automatic recipe generation are performed through the Burner system. In the recipe corpus of this paper, operating system images account for a large proportion of all base images; these operating system images are dominated by Ubuntu and Centos; the sum of Ubuntu and Centos accounts for more than 80% (see Figure 13). Therefore, for the sake of standardization, the base images are uniformly designated as ubuntu: 16.04 and centos: 7. For the same Tag, we used Burner to generate two types of recipes (Singularity Definition File and Dockerfile).

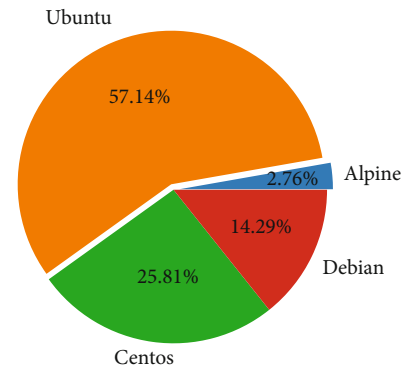


Figure 13: Types and proportions of OS images in recipes.

Table 5: Results of image build test.

Container type	Base image	Build successfully recipes	BSR
Singularity	Ubuntu	20	80%
Singularity	Centos	14	56%
Dockerfile	Ubuntu	18	72%
Dockerfile	Centos	15	60%

As shown in Table 5, in the Singularity Definition File image build test, it was found that 34 of the 50 recipes generated by Burner could successfully build the images; the average image construction success rate reached 68%. Among Singularity recipes, 20 of the 25 recipes with Ubuntu: 16.04 as the base image could successfully build the container images; the image building success rate reached 80%. At the same time, 14 of the 25 recipes with

Table 6: Results of Dockerfile detection with Hadolint.

Rule	Default severity	Result	Description
DL3006	Warning	0	Always tag the version of an image
DL3007	Warning	0	Pin the version explicitly to a release tag
DL3038	Warning	0	Use the-y switch to avoid manual install package
DL3059	Info	0	Multiple consecutive RUN instructions
DL3061	Error	0	Dockerfile must begin with FROM
DL4003	Warning	0	Multiple CMD instructions found
DL3007	Warning	0	MAINTAINER is deprecated
DL3008	Warning	23	Pin package versions in apt-get install
DL3033	Warning	19	Pin package versions in yum install
DL3013	Warning	21	Pin package versions in pip
DL4001	Warning	19	Either use W get or curl but not both
DL3049	Info	50	Label is missing

the Centos: 7 as the base image could successfully build the container images. The difference between the results of Ubuntu and Centos was caused by insufficient Centos recipe samples and limited entity knowledge extracted.

After the image build test, we used the Hadolint tool to detect the Dockerfile, and the results are shown in Table 6. Violations such as DL3006 and DL4000 have been eliminated in the automatically generated Dockerfile; DL3008 and DL3013 have also been improved. The reason of DL3008 and DL3013 cannot be eliminated is that the knowledge of software packages and their versions in the current knowledge base is not sufficient. It can be foreseen that with the enrichment of the knowledge base, DL3006 and DL4000 problems will be improved.

Finally, we analyzed the build logs of the recipes that failed to build, found that the reasons for the failure included environment variable setting errors, missing compilation statements, and "apt-get update" network errors. To solve the above problems, it is necessary to manually further increase the configuration of environment variables and other measures. The highest BSR is 80%, which proves the system can better help users to write recipes, and the Hadolint detect results also prove that the recipes automatically generated by Burner have high quality.

7. Conclusion and Future Work

Compared with the one-phase parsing method, the two-phase parsing method we designed can parse recipes more efficiently. We use the extracted knowledge to build a relatively complete domain knowledge base. On the one hand, this knowledge graph can realize the fine-grained representation of knowledge. On the other hand, the use of graph data and graph algorithms can better solve the problem of dependencies. The automatic generation of recipes using knowledge can greatly reduce the burden of related developers. The recipe automatic generation system Burner can meet the individual needs of different users on the basis of improving the correctness of recipes. The design of Burner revolves around the two core issues of automation and personalization, and the automatic generation of recipes is finally achieved through the construction of knowledge base and the recommendation of entities.

At present, the amount of knowledge in the prototype system is still relatively small, and the dependency inference through this knowledge base may lack version information. In the future, higher-quality recipe generation can be achieved by expanding the scale of the knowledge base. In addition, the software packages in our knowledge base are all officially packaged software (OPS) registered in public repositories and can be installed using package management tools such as apt-get and pip. Some unofficially packaged software (UOPS) cannot be automatically downloaded and installed by package management tools. These UOPS usually need to specify the download address and also need to perform operations such as decompression and switching directory compilation to install. Further research is required for UOPS.

The new versions of Docker and Singularity have added a multi-stage build function, which supports the separation of the compilation environment and the running environment, allowing multiple FROM instructions to appear. This new feature can greatly reduce the size of the final image. We tend to support the multi-stage build function. We will collect the recipe application examples of multi-stage build and improve our research to support the multi-stage build function.

Data Availability

The original recipe dataset and parsing results can be accessed at https://github.com/jhshz520/BurnerRecipe.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Key Research and Development Program of China under grant 2018YFB0204002.

References

[1] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: scientific containers for mobility of compute," *PLoS One*, vol. 12, no. 5, article e0177459, 2017.

[2] D. Nüst, V. Sochat, B. Marwick et al., "Ten simple rules for writing dockerfiles for reproducible data science," *PLoS Computational Biology*, vol. 16, no. 11, article e1008316, 2020.

[3] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[4] P. Souza, G. Kurtzer, C. Gomez-Martin, and P. C. Silva, "Hpc containers with singularity," in *Third EAGE Workshop on High Performance Computing for Upstream*, pp. 1–5, European Association of Geoscientists & Engineers, 2017.

[5] E. Horton and C. Parnin, "Gistable: evaluating the executability of python code snippets on github," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 217–227, IEEE, Madrid, Spain, 2018.

[6] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "Learning from, understanding, and supporting devops artifacts for docker," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 38–49, IEEE, Seoul, Korea, 2020.

[7] E. Horton and C. Parnin, "Dockerizeme: automatic inference of environment dependencies for python code snippets," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 328–338, IEEE, Montreal, QC, Canada, 2019.

[8] S. McMillan, "Making containers easier with hpc container maker," in *Proceedings of the SIGHPC Systems Professionals Workshop (HPCSYSPROS 2018)*, Dallas, TX, USA, 2018.

[9] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pp. 1–10, New York, 2017.

[10] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," in *Proceedings of the Cray User Group*, pp. 33–49, Chicago, 2015.

[11] L. Gerhardt, W. Bhimji, S. Canon et al., "Shifter: containers for hpc," *Journal of physics: Conference series*, vol. 898, article 082021, 2017.

[12] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, "Sarus: highly scalable docker containers for hpc systems," in *International Conference on High Performance Computing*, pp. 46–60, Springer, Cham, 2019.

[13] D. Godlove, "Singularity: simple, secure containers for compute-driven workloads," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, pp. 1–4, New York, 2019.

[14] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An empirical analysis of the docker container ecosystem on github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 323–333, IEEE, Buenos Aires, Argentina, 2017.

[15] G. Schermann, S. Zumberi, and J. Cito, "Structured information on state and evolution of dockerfiles on github," in *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 26–29, IEEE, New York, 2018.

[16] Y. Zhang, G. Yin, T. Wang, Y. Yu, and H. Wang, "An insight into the impact of dockerfile evolutionary trajectories on quality and latency," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 138–143, IEEE, 2018.

[17] "hadolint/hadolint," https://github.com/hadolint/hadolint.

[18] Z. Lu, J. Xu, Y. Wu, T. Wang, and T. Huang, "An empirical case study on the temporary file smell in dockerfiles," *IEEE Access*, vol. 7, pp. 650–659, 2019.

[19] J. Xu, Y. Wu, Z. Lu, and T. Wang, "Dockerfile tf smell detection based on dynamic and static analysis methods," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, pp. 185–190, IEEE, Milwaukee, WI, USA, 2019.

[20] K. Yin, W. Chen, J. Zhou, G. Wu, and J. Wei, "Star: a specialized tagging approach for docker repositories," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 426–435, IEEE, Nara, Japan, 2018.

[21] F. Hassan, R. Rodriguez, and X. Wang, "Rudsea: recommending updates of dockerfiles via software environment analysis," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 796–801, IEEE, New York, 2018.

[22] Y. Chen, J. Kuang, D. Cheng, J. Zheng, M. Gao, and A. Zhou, "Agrikg: an agricultural knowledge graph and its applications," in *International conference on database systems for advanced applications*, pp. 533–537, Springer, Chiang Mai, Thailand, 2019.

[23] Z.-Q. Lin, B. Xie, Y.-Z. Zou et al., "Intelligent development environment and software knowledge graph," *Journal of Computer Science and Technology*, vol. 32, no. 2, pp. 242–249, 2017.

[24] M. Osorio, C. B. Aranda, and H. Vargas, "Dockerpedia: a knowledge graph of docker images and Their Metadata," *International Journal of Software Engineering and Knowledge Engineering*, vol. 32, no. 1, pp. 71–89, 2022.

[25] "An open-source tool from CoreOS designed to identify known vulnerabilities in Docker images," https://github.com/coreos/clair.

[26] V. V. Sochat, C. J. Prybol, and G. M. Kurtzer, "Enhancing reproducibility in scientific computing: metrics and registry for singularity containers," *PLoS One*, vol. 12, no. 11, article e0188511, 2017.

[27] "The container system for secure high performance computing," https://apptainer.org/docs/user/main/.

[28] V. Holen, "A shell script static analysis tool," https://github.com/koalaman/shellcheck.

[29] D. M. German, J. M. Gonzalez-Barahona, and G. Robles, "A model to understand the building and running interdependencies of software," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, pp. 140–149, IEEE, Vancouver, BC, Canada, 2007.

[30] R. Agrawal, T. Imielin'ski, and A. Swami, "Mining association rules between sets of items in large databases," *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, vol. 22, no. 2, pp. 207–216, 1993.

[31] R. Studer, V. R. Benjamins, and D. Fensel, "Knowledge engineering: Principles and methods," *Data & Knowledge Engineering*, vol. 25, no. 1-2, pp. 161–197, 1998.

[32] Y. Park, "The adaptive clustering method for the long tail problem of recommender systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 8, pp. 1904–1915, 2013.

[33] J. Ramos, "Using tf-idf to determine word relevance in document queries," *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1, pp. 29–48, 2003.

[34] J. Lin, H. Pu, Y. Li, and J. Lian, "Intelligent recommendation system for course selection in smart education," *Procedia Computer Science*, vol. 129, pp. 449–453, 2018.