Shipwright: A Human-in-the-Loop System for Dockerfile Repair

Jordan Henkel*§, Denini Silva†§, Leopoldo Teixeira†, Marcelo d'Amorim†, and Thomas Reps*
*University of Wisconsin–Madison, Madison, WI, USA
†Federal University of Pernambuco, Recife, PE, Brazil
{jjhenkel,reps}@cs.wisc.edu {dgs,lmt,damorim}@cin.ufpe.br

***Abstract*—Docker is a tool for lightweight OS-level virtualization. Docker images are created by performing a build, controlled by a source-level artifact called a Dockerfile. We studied Dockerfiles on GitHub, and—to our great surprise—found that over a quarter of the examined Dockerfiles failed to build (and thus to produce images). To address this problem, we propose SHIPWRIGHT, a human-in-the-loop system for finding repairs to broken Dockerfiles. SHIPWRIGHT uses a modified version of the BERT language model to embed build logs and to cluster broken Dockerfiles. Using these clusters and a search-based procedure, we were able to design 13 rules for making automated repairs to Dockerfiles. With the aid of SHIPWRIGHT, we submitted 45 pull requests (with a 42.2% acceptance rate) to GitHub projects with broken Dockerfiles. Furthermore, in a "time-travel" analysis of broken Dockerfiles that were later fixed, we found that SHIPWRIGHT proposed repairs that were equivalent to human-authored patches in 22.77% of the cases we studied. Finally, we compared our work with recent, state-of-the-art, static Dockerfile analyses, and found that, while static tools detected possible build-failure-inducing issues in 20.6–33.8% of the files we examined, SHIPWRIGHT was able to detect possible issues in 73.25% of the files and, additionally, provide automated repairs for 18.9% of the files.**

***Keywords*-Docker, DevOps, Repair**

I. INTRODUCTION

Docker is one the most widely used tools for virtualization. With ∼79% of IT companies using it [1], Docker has made an impact on developers' day-to-day work. Developers use Docker to author images via an artifact called a *Dockerfile*. Images can be based on a variety of operating systems, but primarily, Docker images are Linux-based. Dockerfiles are, effectively, a linear sequence of "setup instructions" that tell the Docker engine how to prepare an image. The final built image is then used to run Docker containers. These containers are similar to lightweight virtual machines. Each container starts with the clean environment specified by its originating Docker image. Together, images and containers allow for isolation, scaling, and reproducibility.

Nonetheless, we found that over 26% of the analyzed samples of Dockerfiles obtained "in the wild" (sourced from GitHub) failed to build successfully. This high a percentage is surprising, because it runs counter to one of the core tenets of Docker, namely, reproducibility. Furthermore, it is outside of the scope of recent efforts to statically analyze Dockerfiles to detect such failures. For example, Hadolint [2] can detect mistakes such as missing or incorrect flags—e.g., forgetting the use of `-assume-yes/-y` when invoking `apt-get install` in a Dockerfile. This flag is required because a Dockerfile build runs without interaction; therefore, forgetting this flag may cause the build to hang. Unfortunately, while Hadolint can detect such a mistake statically, many breakages occur due to a *change in the external environment* and *not a change in the source Dockerfile*. These observations add to the mounting evidence that external changes, such as dependencies changing, can often lead to broken build-related artifacts [3], [4]. As further evidence of this trend, our prototype tool, SHIPWRIGHT, was used to guide 19 accepted pull requests on GitHub and, for each of these patches, the underlying issue was caused by a *change in the external environment* (see Section II-A for an example of one such change).

> **The Problem:** Over a quarter of the GitHub repositories with Dockerfiles that we analyzed had a broken Dockerfile. Current state-of-the-art (static) analysis for Dockerfiles is largely incapable of detecting and/or repairing such broken Dockerfiles.

Given this situation, our work seeks to meet the following high-level goal:

> **Our Goal.** Aid developers in automatically repairing broken Dockerfiles, with the hope of reducing the high percentage of broken Dockerfiles that we observed on GitHub.

A. Contributions

Our work makes the following contributions: 1) *Technique*. We introduce a human-in-the-loop approach to fixing broken Dockerfiles; 2) *Tool*. We made available a tool implementing our technique; and 3) *Data*. We made available an extension of the `binnacle` dataset [5], including build logs.

Technique. Unlike previous approaches that attempt to mine patterns automatically and directly from Dockerfiles, SHIPWRIGHT follows a human-in-the-loop approach to build a *repair* database. We include human supervision to broaden the effectiveness of SHIPWRIGHT: a fully automatic approach would limit the scope of repairs that could be detected. SHIPWRIGHT is designed to act as a "co-pilot" that can sidestep the limitations of a completely automatic approach with

§Equal contributions.

the goal of constructing a comprehensive database of repairs. To build such a database, SHIPWRIGHT uses clustering, human supervision, and a search-based recommendation system we built to leverage vast community-knowledge bases, such as StackOverflow and Docker's community forum. In general, we require repairs to incorporate both some kind of pre-condition (pattern) and a transformation function (patch).

During clustering, one challenge that SHIPWRIGHT needs to address is the heterogeneity of the data, which is a mixture of code and natural language. The mixing of code and natural language makes it non-trivial to design a featurization method for clustering. Therefore, to address this challenge, SHIPWRIGHT uses a modified version of Google's BERT model [6], [7] to *embed Dockerfile build logs*. By using a pre-trained transformer-based neural model, we are able to sidestep tedious feature engineering, and benefit from the diverse corpora on which BERT-based models have been trained. Using this embedding, we perform clustering with `HDBSCAN` [8], in the vector space of embedded build logs, and use the results to cluster failing builds. By using the vectors generated through BERT, we leverage the "understanding" encoded in BERT's language model. To our surprise, we found that recent off-the-shelf language models work well in this domain. Using the generated clusters, we employ human supervision to intuit, for each cluster, a likely root cause of failure and, if possible, engineer one or more automated repairs to save to SHIPWRIGHT's database. Later, SHIPWRIGHT will use this human-generated repair database to attempt automatic repair of failing Dockerfiles.

Tool. The scripts to automate each of the steps of SHIPWRIGHT (see Figure 3) are publicly available at https://github.com/STAR-RG/shipwright.

Data. We have identified a subset of Dockerfiles from the `binnacle` dataset [5] that are amenable to automated builds. (The dataset and filtering criteria are described in Section III.) We have built these files *in-context* (an expensive operation that requires hundreds of hours of compute time), and captured detailed data from the results, including logs from the builds. These build logs represent a significant expansion of the data in the original `binnacle` dataset, and it is our hope that this extended data will accelerate research on diagnostic tools for Dockerfile analysis and repair. This expanded data is available at https://github.com/STAR-RG/shipwright.

B. Evaluation

We evaluated several aspects of SHIPWRIGHT; a summary of our results is as follows: (i) Broken Dockerfiles are prevalent: in the data we analyzed, 26.3% of Dockerfiles failed to build. (ii) Even using optimistic criteria, existing static tools are capable of identifying the cause of a failure in only 20.6%–33.8% of the broken Dockerfiles. (iii) SHIPWRIGHT is capable of clustering broken Dockerfiles and offering actionable solutions: for files that clustered, SHIPWRIGHT provides automated repairs in 20.34% of the cases; for files that did not cluster, SHIPWRIGHT is still able to provide automated repairs in 18.18% of cases. (iv) In a "time-travel" analysis, we found that SHIPWRIGHT would be able to provide actionable solutions to 98.04% of the Dockerfiles that we found initially broken and then subsequently fixed in their respective repositories. Finally, we used the reports from SHIPWRIGHT to submit 45 Pull Requests to still-broken Dockerfiles. Of these, 19 have been accepted. These results provide initial, yet strong, evidence that SHIPWRIGHT is useful to help developers fix broken Dockerfiles.

II. SOURCES OF BUILD FAILURES

This section describes some of the distinct sources of problems that can lead to build failures in Dockerfiles.

A. Breaking Changes in External Files

This kind of failure occurs when a Dockerfile's external dependencies (such as the file's base image or URLs embedded within the file) are changed; often these changes external to the Dockerfile will require a change in the Dockerfile itself. To illustrate this problem, consider the case where the developer used `latest` to indicate the version of the base image of her Dockerfile, as in `FROM ubuntu:latest`. These base images are downloaded from Docker Hub [9], a distributed database that is part of the Docker ecosystem. The problem with using `latest` is that a change to the base image may require changes to the Dockerfile. Unfortunately, there is no clear way to incorporate those required changes automatically. For instance, the `python-pip` package is part of Python 2, and Python 2 is unavailable on Ubuntu images higher than 18.04. Consequently, a build on a Dockerfile with the command `apt-get -y install python-pip` will pass when the file is based on Ubuntu images 18.04 and lower, but it will fail on higher versions, including the latest LTS version of Ubuntu.

We used the SHIPWRIGHT toolset to analyze and cluster hundreds of broken Dockerfiles, looking for common error patterns in their build logs and associated Dockerfiles. Using a human-in-the-loop process, we then extracted patterns and associated them with candidate repairs. For example, when running the command `docker build` on the Dockerfile `FROM ubuntu:latest...RUN apt-get -y install python-pip...`, Docker reports the message "*Unable to locate package python-pip*" on output. When that message is present in the logs, we found that the typical image in Dockerfiles is Ubuntu and the version is either undefined, `latest`, or `20.04`. We also noticed that this error message appears not only with `python-pip`, but with other packages as well.

We expressed these patterns with regular expressions to be checked against the Dockerfile (the static data) and error logs (the dynamic data). For instance, we expressed the pattern for the problem above with the regex "`FROM ubuntu`(ϵ | :latest | :20.04)" $\wedge$ "*Unable to locate package* (.*)" $\in$ log. Note that such an expression 1) defines a pattern over the Dockerfile, 2) defines a pattern over the output log, and 3) uses the groups "(ϵ...)" and "(.*)" to bind data to variables for later use.

Figure 1 shows two possible solutions to the problem above. The first solution is to fix the base image to the most recent

```
# solution 1, use version 18.04
FROM ubuntu:18.04
RUN apt-get -y install python-pip
... #remaining code

# solution 2, manually install the package
FROM ubuntu:latest
ARG DEBIAN_FRONTEND=noninteractive
RUN apt-get -y install python2 curl software-prop... \
  && add-apt-repository universe \
  && curl https://.../get-pip.py --output get-pip.py \
  && python2 get-pip.py
... #remaining code
```

Fig. 1: Solving `python-pip` unavailable on `ubuntu:latest`.

version with which the command can still be executed. The following abstract operation characterizes the repair: replace `$0` with `:18.04`. The symbol `$0` refers to the regex group matching the Ubuntu image in the Dockerfile (i.e., "(ϵ...)") that must be replaced with one specific Ubuntu version (e.g., `:18.04`). The second solution is to install Python 2 and its toolset. The following operation characterizes the repair: add `ARG DEBIAN_FRONTEND=...` after "`FROM ubuntu`(ϵ | `:latest` | `:20.04`)". The symbol "..." is a placeholder for the text associated with the second solution from Figure 1.

SHIPWRIGHT records the association between a given pattern (of build error) and possible solutions, such as the two repairs above. From this information, SHIPWRIGHT is able to repair broken Dockerfiles whose build logs match some of the pre-recorded patterns. Table I describes this example under the row with "Id" 5. Note that the repair operation consists of multiple possible solutions, hence the comma and dots, as we only show the first solution. In this case, SHIPWRIGHT produces two versions of the Dockerfile and the developer should choose which one suits best their needs. We elaborate on SHIPWRIGHT's workflow in the following sections.

It is worth noting that prior work has investigated the impact of breaking changes in package-management systems (e.g., in the Linux package manager, npm, maven, etc.) [10]–[14]. SHIPWRIGHT is not restricted to this kind of issue, and is distinct from prior work on the application context and solution used. Section VII elaborates on related work.

B. Inconsistent Version Dependency Within Project

This kind of failure occurs when there is an inconsistency between the versions of a Dockerfile and some of the files it references within the project. Figure 2 shows a concrete example that illustrates the problem. The Dockerfile requires an image for Ruby version 2.6.3, whereas the application code declares a dependency on a newer Ruby version (2.6.5) [15]. The execution of the command `RUN bundle install` triggers an error, producing the following message on output "Your Ruby version is 2.6.3, but your Gemfile specified 2.6.5". In this case, the solution was to replace line 1 of this Dockerfile with `FROM ruby:2.6.5`. The pattern and corresponding repair explained above are listed in Table I under row "Id" 6. A similar repair is "Id" 1, which is also related to Ruby.

```
FROM ruby:2.6.3
RUN apt-get update -qq && apt-get install -y \
...
RUN gem install bundler:2.0.1
RUN bundle install # <-- Gemfile depends on ruby 2.6.5!
ADD . /app
```

Fig. 2: Inconsistent Ruby version dependencies.

C. Missing Commands in the Dockerfile

This failure occurs when a given Dockerfile uses a command that is unavailable on a given image. The solution in that case is to install the command using the proper syntax, because it depends on the version of the image. Table I lists one example of this error pattern and the respective repair under row "Id" 8.

D. Project-Specific Failures and Suggestions

We observed that many broken Dockerfiles require repairs that are project-specific and cannot be generalized. In those cases, SHIPWRIGHT is unable to produce a repair to the broken file. Instead, in those cases, SHIPWRIGHT provides *suggestions*. For instance, consider the case where a Dockerfile includes a command to deploy a Node.js server, such as `RUN npm run build`. The execution of that command fails because there is an error in the Node.js project. There is nothing to fix within the Dockerfile. The developer needs to analyze what is wrong in her Node.js project and fix it. In that case, SHIPWRIGHT reports a suggestion, such as "NPM build error..". As another example, consider a case in which a command refers to a broken link, such as `RUN wget <url>`. SHIPWRIGHT cannot guess how to fix the broken link. Table II shows a sample of suggestions provided by SHIPWRIGHT. In the future, it would be interesting to examine how one might combine SHIPWRIGHT with other tool-specific and language-specific repair techniques. Tools that have a *combined understanding* of DevOps artifacts and the programs these artifacts support represent an intriguing area for future work.

III. DATASET

We use an expanded version of the `binnacle` dataset [5] as the source of Dockerfiles to analyze. The `binnacle` dataset consists of *all* Dockerfiles from GitHub repositories *with ten or more stars*. These Dockerfiles represent a broad and largerly unfiltered picture of the state of Dockerfiles one might find in popular GitHub repositories. Although the original dataset was created in 2019, the `binnacle` toolchain allows us to capture recent data using the same methodology; thus, we populated our dataset with more recent data (June 2020) extracted using the same tools. Unfortunately, directly using the Dockerfiles in this dataset is challenging for two reasons: (i) many Dockerfiles in the dataset come from the same repository and, in such cases, the *purpose* of the Dockerfiles is obscured, making efforts—like automated builds—more difficult; (ii) many Dockerfiles are nested deep within repositories (especially when repositories contain many independent projects or services). In either case, automated builds are challenging because the *intent* behind the Dockerfile

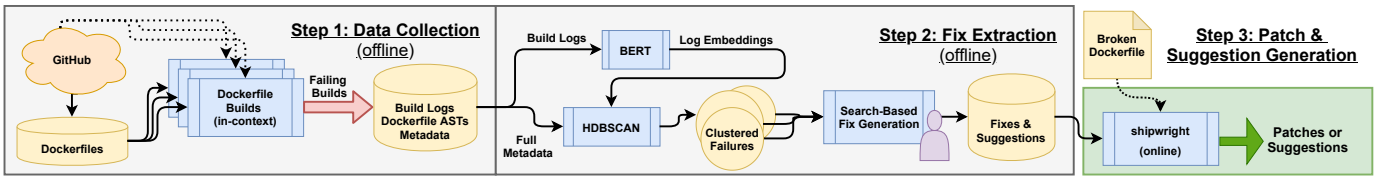

Fig. 3: SHIPWRIGHT's 3-step workflow. In step (1), a database of Dockerfiles and GitHub metadata is used to perform *in-context* builds. The results of these builds are stored in a local database along with various forms of metadata. In step (2), SHIPWRIGHT uses BERT and HDBSCAN to cluster build data [6]–[8]. The clusters are then fed to SHIPWRIGHT's search-based repair-generation and suggestion-generation process. During this step, SHIPWRIGHT, with the assistance of a human, builds a database of repairs and suggestions. Finally, in step (3), online usage begins: new files are fed to SHIPWRIGHT and, if matching database entries are found, SHIPWRIGHT provides relevant repairs or suggestions.

is difficult to infer. In case (i), it is difficult to infer which of the (many) Dockerfiles should be built. In case (ii), it is difficult to infer an appropriate *context directory*, which is a pre-requisite to building a Docker image.

Dataset Filtering. To address these issues, we filtered the files from the `binnacle` dataset using two criteria: (a) we only considered repositories with *a single Dockerfile*, and (b) we required that the given Dockerfile resides within the repository's *root directory*. For such a repository, it is not unreasonable to assume two things: (i) the Dockerfile is intended to produce an artifact corresponding to the given repository (because it is the *only* Dockerfile in that repository), and (ii) the Dockerfile likely uses the repository's root directory as its *build context*: the Dockerfile resides in the root directory, and the `docker build` command assumes, by default, that the target Dockerfile resides within the given context directory. Performing this filtering yielded 32,466 repositories and corresponding Dockerfiles that may be amenable to automated builds. It is this subset of the original dataset (a refreshed version of `binnacle`'s dataset) that we used in our studies.

Building Dockerfiles at Scale. SHIPWRIGHT performed *in-context* builds on 20,526 of the 32,466 Dockerfiles in our *filtered* dataset (recall: we filter to find Dockerfiles from the original dataset that are likely amenable to *automated* builds). Although we would have liked to use all 32,466 Dockerfiles, we encountered some problems performing in-context builds on that many files in a reasonable time frame, which forced us to use a smaller set of 20,526 files. We tried various approaches to scaling these in-context builds, but distributing this kind of process would require Docker installations across a wide variety of machines—in practice, this requirement was difficult to satisfy: none of the distributed-computing resources we had access to would allow us to control a *Docker daemon* on the distributed machines (while, for many distributed platforms, running *containers* is easy). This requirement, while understandable from a security perspective, forced us to run builds in a non-distributed way (on a single large server). Given this constraint, some builds either *time out* (we set a limit of 30 minutes) or, due to contention from running multiple builds on this single server and daemon instance, some builds fail to complete due to errors internal to the Docker daemon; builds that fail in this manner are marked as *undetermined*. Instead of revisiting undetermined builds, we spent our resources on building a larger portion of our filtered dataset. Through this process, we captured the results from 20,526 Dockerfile builds, and saved these results to a database for further analysis. Section IV-A describes this offline data processing with the SHIPWRIGHT toolset in further detail.

IV. SHIPWRIGHT

Figure 3 shows the workflow of SHIPWRIGHT as a pipeline of three steps, organized according to their respective goals.

The goal of the first step is to analyze a corpus of broken Dockerfiles—mined from GitHub—and to perform *in-context* builds so that logs can be acquired (Section IV-A). The goal of the second step is to cluster broken Dockerfiles and find repairs (i.e., transformation functions on Dockerfiles) (Section IV-B). Given a cluster, SHIPWRIGHT automatically elaborates search queries from log files of representative Dockerfiles within the cluster. A human then supervises the creation of repairs and suggestions by (i) looking for error patterns as manifested in existing QA forums resulting from the search query, and (ii) creating plausible repairs (or, if no automated repairs are possible, creating suggestions) which are saved in a database for later, online, use. Finally, the third (interactive) stage of the pipeline looks for actual repairs for a broken Dockerfile (Section IV-C). This component takes as input the output produced in the previous (offline) stages and a broken Dockerfile, and produces either (i) an automated repair, (ii) a suggestion (in cases where repair cannot be automated), or (iii) an indication that no existing repairs or suggestions apply. The following sections describe each step in detail.

A. Data Collection

This component of SHIPWRIGHT builds and analyzes Dockerfiles from a pre-existing corpus. We utilize the 32,466 Dockerfiles described in Section III as our input corpus. Those files were filtered to be amenable to automated builds. For each file in this corpus, SHIPWRIGHT does the following: 1) *Clones* the originating repository for the given Dockerfile into a unique `/tmp/<repo-id>` directory; 2) *Runs* `docker build -f <Dockerfile> /tmp/<repo-id>`, which builds a `<Dockerfile>` from our dataset using the root directory of the cloned repository as the build context. Building *in context* is crucial because the build may need to access files from the originating repository to complete successfully. Although we are interested in build failures, we want to avoid trivial failures; 3) *Discards* builds that still have trivial failures; and 4) *Saves*, for each failing build, execution logs (standard output and standard error streams), the AST for the given Dockerfile, and various metadata (e.g., repository information, image history,

Fig. 4: **Clustering Example.** Starting with several broken Dockerfiles, SHIPWRIGHT clusters by extracting standard error logs, applying aggressive token splitting (we split on snake and camel case, as well as several operators that may occur in code snippets) and string normalization (we lowercase the input, clear large blocks of repetitive characters, like extraneous white space, and we strip certain special characters/unicode), and passing the resulting sequences to BERT. BERT takes the input sequences and produces vectors (shown as points in a high-dimensional space). SHIPWRIGHT uses that mapping, from failing build logs to points in a vector space, along with a clustering algorithm (HDBSCAN), to produce its clusters, shown in the bottom right of the figure. The final clustering process has the advantage of being *semantics aware*: BERT understands some of the nuances of the English language, and our clustering benefits from this capability.

and a log of the git clone procedure). An example would be a broken build caused by an execution failure in a directive, such as `COPY` or `ADD`. Although builds are performed *in context*, it is still possible that the Dockerfile is intended to be built as part of a more complex workflow with a context directory that is different from the repository root directory. Unfortunately, this information may exist in some third-party script, or may be user-supplied.

Figure 3 illustrates this workflow in the box named "Step 1: Data Collection." We note that data collection is quite costly: we ran a 32-core CentOS workstation for several weeks and, during that time, managed to build about two thirds (20,526) of the 32,466 files in our dataset. (See Section III for a discussion of the difficulties of building many Dockerfiles at scale.) Although builds can be parallelized, there is only one Docker daemon per installation of Docker—this situation creates a limit to the practical concurrency that can be achieved, along with the network bandwidth to the workstation used for analysis. We attempted to perform builds on a high-throughput cluster but, unfortunately, the strict security requirements of such clusters prevent deploying a workflow involving Docker images builds, which effectively execute untrusted code.

B. Repair & Suggestion Extraction

This step of SHIPWRIGHT works as follows. First, it uses HDBSCAN, applied to embeddings,[1] to partition the Dockerfile data produced in the previous step. Second, it uses those clusters to assist a human with the task of searching for solutions and building a database of repairs. We now elaborate on each of these steps.

Clustering: SHIPWRIGHT attempts to cluster failing Dockerfile builds using embeddings and `HDBSCAN` (a hierarchical variant of `DBSCAN`, a classic clustering algorithm [16]). The difficulty of clustering in this domain is two-fold: (i) the data to cluster is heterogeneous, and is often a mix of code and natural language (i.e., the build logs, which will often contain a description of the failure in English and a reproduction of the Bash or Dockerfile snippet that leads to the error), and (ii) although we would like to cluster on the *cause* of build failures, we do not have a way to definitively extract the cause of a given build failure; therefore, we must use data that is, at best, a proxy or symptomatic of the root cause of failure. In particular, we use a tokenized version of the build logs for the failing build (which may include a variety of things: debug output, warnings, and errors—some of which may include code snippets) as input to BERT to produce embeddings.

Despite these challenges, SHIPWRIGHT is able to perform clustering by leveraging a key insight: recent off-the-shelf language models, such as BERT, GPT-2 and, recently, GPT-3, have reached impressive levels of sophistication [7], [17], [18]. Given the inputs these models are trained on (roughly, massive crawls of the internet), it is highly likely that such models have seen websites like StackOverflow, which mix both natural language and code. Therefore, to address challenge (i) (the heterogeneous mix of code and natural language), SHIPWRIGHT leverages a sufficiently sophisticated off-the-shelf language model (BERT),[2] to obtain *embeddings*. In particular, SHIPWRIGHT uses a variant of BERT suited to the task of sentence embedding, in which similar sentences should end up "close" in the embedding space. SHIPWRIGHT applies this BERT variant to the last few lines of the error logs to produce a vector representation of each broken Dockerfile. These vectors are then fed to HDBSCAN, which produces clusters. Figure 4 illustrates the SHIPWRIGHT clustering process.

Searching for Repairs or Suggestions: This component of SHIPWRIGHT takes a set of clusters as input, and produces a *list of pairs* as output. The first element of the pair is a signature that identifies the issue (in the Dockerfile and its logs) whereas the second element of the pair is either (i) a repair, consisting of a pure transformation function that takes a Dockerfile as input and produces another file as output, or (ii) a suggestion (about what needs to be repaired and how) for the cases where human knowledge is necessary to prepare the repair. We elaborate on each of these two cases in the following and Tables I and II provide examples of such pairs.

Case 1 (Searching for Repairs): SHIPWRIGHT uses a

[1]Embeddings refer to high-dimensional vectors of numbers that are used as a proxy for non-numeric artifacts (such as text or code). Embeddings are often of use, because many operations can be performed in the resulting vector space, and later mapped back to the originating artifacts.

[2]Ideally, one would try GPT-3 because it is the newest and largest such language model; unfortunately, for now, access to GPT-3 is quite limited, and we were unable to obtain access.

search-based recommendation system to assist a human in locating repairs of broken Dockerfiles. SHIPWRIGHT proceeds as follows: it selects a cluster and a representative Dockerfile from that cluster; it extracts keywords from the logs of that file; it builds a search string from those keywords; it submits the corresponding query to a search engine; it filters the outputs from related community forums; and it reports a list of the top-5 URLs as output for a human to inspect. Human inspection consists of reading proposed solutions on discussion forums, and then applying a given solution to the representative Dockerfile from the cluster. If that solution is plausible, i.e., if it allows the Dockerfile to build an image successfully, the next step is to check if the error-pattern/repair-function pair is applicable to other Dockerfiles in the cluster. While doing so, the human inspector looks for opportunities to generalize the pattern and repair function to avoid overfitting a solution to a particular case. For instance, in the example given in Section II-A, the initial solution was too narrow, focusing on fixes of files containing the exact message "`Unable to locate package python-pip`" in the output log. However, we observed similar error messages, referring to different packages. In this case, the solution was to replace "`python-pip`" in that string with a symbolic name for a package. To sum up, SHIPWRIGHT leverages community knowledge bases (e.g., StackOverflow and Docker's community forums) to find solutions to known issues, such as those presented in Section II.

SHIPWRIGHT supports a total of 13 repair patterns. Table I shows the pairs of (1) build-error signatures—referred to as a pattern—and (2) a corresponding repair for 10 of them. Column "Id" shows the id of a pair. Column "Pattern" shows the error pattern, which is a regular expression that matches a string in the error logs (the dynamic part) and/or a string in the Dockerfile (the static part). Column "Repair" shows a function, in natural language, describing how to transform and fix a broken Dockerfile. We use the keywords add, remove, and replace to describe operations that need to be performed on the Dockerfile. We informally described the semantics of these operations with examples in Section II-A. Although there is no fundamental reason preventing us from creating these transformations automatically, we wrote the code implementing these transformation functions because we found empirically that creating these functions was not a time-consuming error-prone task. Finally, column "Src." shows a reference for the solution on the web.

Case 2 (Searching for Suggestions): There are cases where SHIPWRIGHT cannot produce a repair. For example, a Dockerfile whose build fails because of a compilation error or broken URL requires a human to fix the underlying error. For those cases, we report a suggestion, i.e., generic advice on what needs to be done. Table II shows a small sample from the total of 50 suggestion patterns that SHIPWRIGHT supports (in addition to 13 repair patterns). Column "Id" shows the id of the suggestions, column "Pattern" shows the signature, and column "Suggestions" shows the suggestion message.

C. Repair and Suggestion Generation

SHIPWRIGHT can be used to repair Dockerfiles or provide suggestions using the database generated in step 2 (Section IV-B). Given a Dockerfile, SHIPWRIGHT iteratively examines the repairs and suggestions and, given a match, it either (i) produces a patched file, by applying a repair, or (ii) provides a suggestion message to the user. If neither a repair nor a suggestion with a matching pre-condition exists within the database, SHIPWRIGHT is still able to use its search-based process to guide a human in producing fixes and suggestions, as we did in step 2 (Section IV-B). This search-based process provides a user with a small set of (filtered) links to resources likely to help in fixing the given input file. In summary, SHIPWRIGHT, during step 3, produces either: (i) a Dockerfile repair, (ii) a suggestion on how to fix the broken build, or (iii) a curated set of results from a search-based process that may provide solutions to the underlying build issue.

V. EVALUATION

The goal of SHIPWRIGHT is to help developers fix broken Dockerfiles. It does that through a combination of (i) clustering of broken Dockerfiles (by likely root cause), and (ii) a search-based method to find repairs (and, if no automated repairs are feasible, suggestions). To gain insights into the landscape of broken Dockerfiles used in GitHub projects and to understand SHIPWRIGHT's efficacy, we pose the following research questions.

RQ1. *How prevalent are Dockerfile build failures in projects that use Docker on GitHub? Can existing (static) tools identify the failure-inducing issues within these broken files?*

Rationale: The purpose of this question is to evaluate the potential impact of SHIPWRIGHT. If build failures are rare, then impact is limited. Furthermore, reproducibility is a core tenet of Docker—it would be surprising to find many broken Dockerfiles. We also assess the ability of existing (static) tools to identify issues that may lead to failing Dockerfile builds.
Metrics: We used the following metrics to answer RQ1: 1) the fraction of Dockerfiles in our dataset with builds that fail; 2) the relationship between failures and project popularity; and 3) the success rate of existing (static) tools in predicting Dockerfile build failures. The first metric evaluates the fraction of Dockerfiles that we mined from GitHub that fail to build because the Dockerfile is broken (for non-trivial and non-toolchain-related reasons). The second metric examines the relationship between the number of GitHub stars a given repository has (a common proxy for popularity on GitHub) and whether that repository contains a broken Dockerfile. This measurement helps us ascertain whether popular repositories suffer from broken Dockerfiles at the same rate as less popular repositories. Recall that we do not have any Dockerfiles from repositories with less than 10 GitHub stars (Section III). Finally, the third metric ascertains the ability of pre-existing tools, namely, Hadolint [2] and `binnacle`'s rule checker [29], to find issues within broken Dockerfiles.

TABLE I: Selected Repairs.

Id	Pattern	Repair	Src.
1	"*ERROR: Error installing bundler* ∈ log ∧ "*bundler requires Ruby version* ≥ ([0-9]+.[0-9]+.[0-9]+)" ∈ log	replace `FROM ruby:(.*)` with `FROM ruby:$0`	[19]
2	"*Rpmdb checksum is invalid: dCDPT*" ∈ log	add `RUN yum install -y yum-plugin-ovl` after `FROM(.*)`	[20]
3	"*E: Some index files failed to download. They have been ignored, or old ones used instead*" ∈ log	replace base image with latest release from hub.docker.com	[21]
4	"*E: Package 'libpng12-dev' has no installation candidate*" ∈ log	replace `libpng-12dev` with `libpng-dev`	[22]
5	"`FROM ubuntu`(∈|:latest|:20.04)" ∧ "*Unable to locate package* (.*)"∈ log ∈ Dockerfile	replace `$0` with `:18.04`, ...	[23]
6	"*but your Gemfile specified* ([0-9|\\\.]+)" ∈ log ∧ "`FROM ruby`(.*)" ∈ Dockerfile	replace `FROM ruby:(.*)` with `FROM ruby:$0`	[24]
7	"*invalid byte sequence in US-ASCII*" ∈ log ∧ "`FROM ruby`(.*)" ∈ Dockerfile	add `ENV LANG C.UTF-8` after `FROM(.*)`	[25]
8	"*sh:* (.*)*: not found*" ∈ log	**if** "`FROM alpine`(.*)" ∈ Dockerfile **then** add `RUN apk add -no-cache $0`. **else** add `RUN apt-get -y update && apt-get -y install $0`	[26]
9	"*ERROR: unsatisfiable constraints: bzr (missing):*" ∈ log ∧ "`FROM alpine`(.*)" ∈ Dockerfile	remove `bzr` in "`apk add`" command	[27]
10	"*conda: not found*" ∈ log ∧ "`RUN curl` (*https://repo.continuum.*(.*))" ∈ Dockerfile	add `-L` before `$0`	[28]

TABLE II: Selected Suggestions.

Id	Pattern	Suggestion
1	"*mix*" ∈ log ∧ "*Code.LoadError*" ∈ log	Problem running mix on the Elixir project...
2	"*tsc*" ∈ log ∧ "*: error TS*" ∈ log	Error during TypeScript compilation with tsc. Please check your .ts files.
3	"*wget: server returned error*" ∈ log ∨ "*wget: unable to resolve host address*" ∈ log	`wget` error, you have a broken URL. Please check the log.
4	"*npm ERR!*" ∈ log	NPM Error, check your files, in particular, "npm install" commands.
5	"*curl:*([0-9]+).*" ∈ log	`curl` error, you have a broken URL. Please check the log.

RQ2. *Can we use off-the-shelf language models, like BERT, to easily cluster broken Dockerfiles?*

Rationale: Given the number of observed failures, it is reasonable to ask whether many failures are *unique*. If many failures are similar, one might hope that generalized repairs exist. Furthermore, if failing Dockerfile builds can be clustered, those clusters may be used to bootstrap finding repairs. Finally, if we can leverage the level of understanding available in large off-the-shelf language models (like BERT), then we can design robust clustering routines with little specialized engineering effort, and avoid techniques based on manually designed heuristics.

Metrics: To answer RQ2, we examine the percentage of clusters (generated using SHIPWRIGHT), where all elements share a single (likely) root cause. This metric provides insight into SHIPWRIGHT's ability to cluster broken Dockerfiles and the usefulness of those clusters—good clustering allows for finding multiple exemplars for a single failure which, in turn, makes the task of generating automated repairs simpler.

RQ3. *How effective is* SHIPWRIGHT *in producing repairs? (i) To what extent do repairs cover the failures from our dataset? (ii) For failures that can be clustered, is it possible to generalize repairs? (iii) What can be done for failures from non-clustered files?*

Rationale: The purpose of this question is to evaluate SHIPWRIGHT's effectiveness on our dataset. If proposed solutions are unable to cover a variety of Dockerfiles, then SHIPWRIGHT's usefulness is questionable.

Metrics: 1) We measured the fraction of broken Dockerfiles (from our dataset) for which SHIPWRIGHT produces a repair. 2) For the *set of clusters* that SHIPWRIGHT produces, we measured the extent to which repairs generalize. For that, we measure "coverage" (i) in the cluster that originated that pattern, and (ii) across different clusters. Coverage refers to the portion of elements within a cluster that match the same pre-condition for a repair. 3) For broken Dockerfiles that *did not cluster*, we measured how often SHIPWRIGHT provides a repair. In all cases, we also evaluated SHIPWRIGHT's ability to provide *suggestions* if repairs were not feasible. Collectively, these metrics measure how effective SHIPWRIGHT is in proposing solutions to the broken files in our dataset.

RQ4. *How effective is* SHIPWRIGHT *in reducing the number of broken Dockerfiles in public repositories?*

Rationale: Although RQ3 seeks to evaluate SHIPWRIGHT's ability to fix broken Dockerfiles, there still remains a question of SHIPWRIGHT's usefulness in practice. RQ4 seeks an understanding of SHIPWRIGHT's effectiveness to meet our overarching goal: fixing broken Dockerfiles in public repositories.

Metrics: To answer RQ4, we used two metrics: 1) What proportion of Dockerfiles that appear in our dataset as broken Dockerfiles, but have since been fixed, *would also have been fixed, had we applied* SHIPWRIGHT? 2) How often can we

use SHIPWRIGHT to produce pull-requests that are accepted by external reviewers? The first metric refers to a kind of "time-travel" analysis because, using updates that took place during the period in which we built SHIPWRIGHT, we can attempt to measure how successful we *would have been* had SHIPWRIGHT existed at an earlier date, and had we applied it. Nevertheless, this metric is still a "simulated" one. Therefore, the second metric quantifies SHIPWRIGHT's "real-world" applicability by actually using it to produce repairs and submitting them for (external) review.

A. Answering RQ1: How prevalent are Dockerfile build failures in projects that use Docker on GitHub? Can existing (static) tools identify the failure-inducing issues within these broken files?

To answer RQ1, we used SHIPWRIGHT to build a random sample of Dockerfiles from our (filtered) dataset. In total, we tried to build 20,526 Dockerfiles and found 5,405 broken Dockerfiles. This gives us an estimated 26.3% "breakage rate" for Dockerfiles in our overall dataset. The large amount of broken Dockerfiles on GitHub runs counter to one of the core reasons for using Docker: *reproducibility*. Aside from broken Dockerfiles, we encountered 393 Dockerfiles with builds that time out (we use a threshold of 30 minutes) and 3,514 Dockerfiles with undetermined results (which arise due to the pressure that multiple concurrent builds place on the Docker daemon). Neither timeouts nor builds with undetermined results are counted as broken Dockerfiles. Instead, we count these results as successful builds to give a conservative estimate (and lower bound) of the "breakage rate" for Dockerfiles in our dataset. Figure 6 provides a visual overview of these categories.

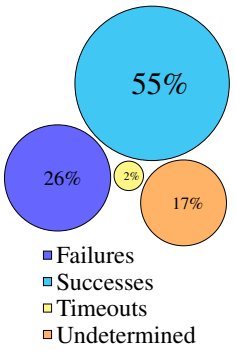

Fig. 6: Breakdown of the 20,526 files we attempted to build.

To put these results in context, we also examined the distribution of stars for the 5,405 repositories in our dataset. For these repositories, we find that: (i) a third have 18 stars or fewer, (ii) a third have greater than 18 stars, but fewer than 51 stars, and (iii) a third have 51 or more stars. This distribution was surprising, especially because some repositories with broken Dockerfiles had many thousands of stars. We spot-checked some of these cases and found that, indeed, even quite popular repositories can have broken Dockerfiles. For example, the MEAN stack project [30] has over 12K stars, yet it contains a Dockerfile that fails to build.

Finally, we also tested the capabilities of two existing (static) tools: `binnacle` [29] and Hadolint [2]. For both tools, we sought an estimate of the number of broken Dockerfiles for which each tool *identifies a possible build-breaking issue*. Because we found it impractical to manually examine the tools' outputs on each of the 5,405 broken files, we instead used a (generous) estimate based on how often each tool reports a rule violation for an issue that *might cause a build to break*. For example, Hadolint can identify when the version of an image used for a base in a Dockerfile is un-pinned; thus, if Hadolint reports a rule violation in this category, on any file, we count it as Hadolint identifying a *possible build-breaking issue* (and mark the file as "solved" by Hadolint). In total, Hadolint identifies such issues in only 33.8% of files, and `binnacle` identifies such issues in only 20.6% of files.

> **Summary of RQ1:** The presence of broken Dockerfiles on GitHub is common. Furthermore, even highly starred repositories sometimes contain broken Dockerfiles. Finally, existing static tools only identify plausible build-breaking issues in 20.6–33.8% of cases (and, even when issues are identified, such tools do not provide repairs).

B. Answering RQ2: Can we use off-the-shelf language models, like BERT, to easily cluster broken Dockerfiles?

To generate the clusters we use throughout our evaluation, we first performed a grid search against SHIPWRIGHT's clustering algorithm. During this search, we focused on exploring the space of hyperparameters used in HDBSCAN—the embeddings, although generated by a neural model, are not "tunable" without investing in re-training the model, which is outside the scope of SHIPWRIGHT. We searched approximately 200 configurations and found, on average, HDBSCAN was able to cluster 34% (1,836) of the 5,405 broken Dockerfiles identified by SHIPWRIGHT. In the clustering that we used, consisting of 144 clusters containing 1,814 files, we were able to confirm that 36.5% of the clusters consisted of Dockerfiles that all had the same root cause for their failures.

> **Summary of RQ2:** SHIPWRIGHT's approach to clustering Dockerfiles can, on average, cluster 34% of our dataset, and, for over a third of the clusters generated, we can confirm that a *single* issue covers all failing Dockerfiles within a cluster.

The answer to RQ2 bodes well for using clusters to bootstrap finding automated repairs. However, we note that the clustered files only make up a portion of broken files: therefore, to assess generalizability, RQ3 examines SHIPWRIGHT's ability to use repairs learned from our clustered files and apply them to non-clustered files.

C. Answering RQ3: How effective is SHIPWRIGHT in producing repairs? (i) To what extent do repairs cover the failures from our dataset? (ii) For failures that can be clustered, is it possible to generalize repairs? (iii) What can be done for failures from non-clustered files?

This question evaluates SHIPWRIGHT's effectiveness on our dataset of broken Dockerfiles (Section III).

RQ3.1 evaluates how much of the set of broken Dockerfiles can be addressed with the repairs that SHIPWRIGHT generates. Figure 5 shows the effects of the repairs (and suggestions) that we found across the 144 clusters produced by SHIPWRIGHT. Each vertical bar denotes one cluster. These bars are divided into three segments. The size of the segment at the bottom of

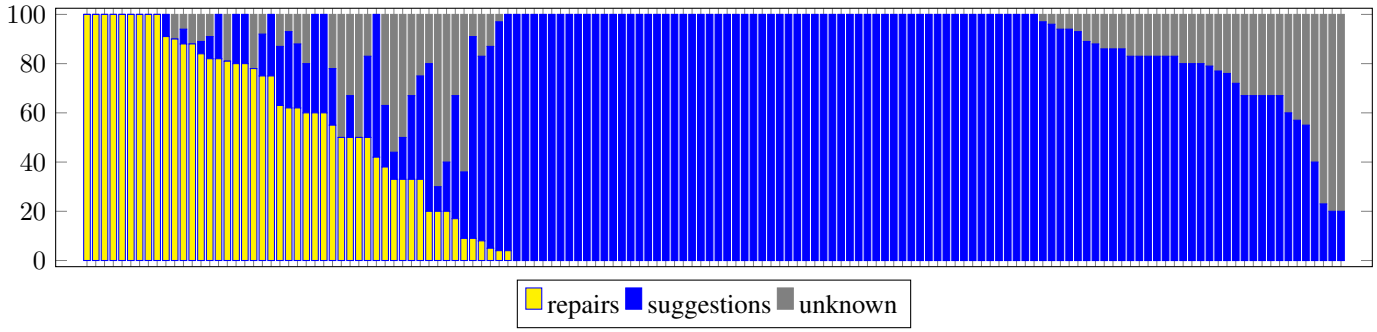

Fig. 5: Proportion of different kinds of solutions within each cluster (excluding singleton clusters).

the bar (in yellow) represents the percentage of failures in a given cluster for which SHIPWRIGHT provided an automated repair; the size of the segment in the middle of the bar (in blue) represents the percentage of failures for which SHIPWRIGHT provided suggestions (which are generated in cases where no repairs apply); and the size of the segment at the top of the bar (in gray) represents the percentage of failures for which SHIPWRIGHT could *not* find a solution.

Summary of RQ3.1: The 13 repairs created with SHIPWRIGHT offered solutions to 20.34% of the 1,814 broken and clustered Dockerfiles. In cases where no repairs were applicable, SHIPWRIGHT's 50 suggestions applied to an additional 69.63% of the broken and clustered Dockerfiles.

RQ3.2 evaluates the ability of the repairs to generalize to a large number of cases. Table III shows the relative amount of broken Dockerfiles that each one of our 13 repair patterns covered.

TABLE III: Repair Coverage.

Id	# Clusters	Coverage (%)	
		Parent	Average
1	4	100	61
2	3	40	25.67
3	8	100	54
4	3	60	49.34
5	2	88	88
6	2	100	90.50
7	1	82.14	82.14
8	6	100	88.67
9	1	100	100
10	2	100	95.5
11	3	62	50
12	2	80	42
13	3	80	60

Column "Id" refers to the id of the repair (most of which listed in Table I), and column "#Clusters" shows the number of clusters where the corresponding repair could fix at least one of the broken Dockerfiles in it. Error patterns are extracted from a given cluster, which we refer to as "parent". Column "(Coverage) Parent" then shows the fraction of broken Dockerfiles within the parent cluster that were corrected using the respective repair. Column "(Coverage) Avg." shows the average fraction of repaired files across the different clusters affected by a repair pattern.

Summary of RQ3.2: The 13 repair patterns produced with SHIPWRIGHT generalized well within the parent cluster (avg. 84.01%) and across affected clusters (avg. 68.22%).

Recall that a total of 3,586 of the 5,400 broken Dockerfiles (66.4%) were *not* clustered. For non-clustered files, SHIPWRIGHT produced repairs to 18.18% of them. Overall, SHIPWRIGHT produced an actionable solution to the developer in 64.81% of the files that were not associated with any cluster (18.18% from repairs and an additional 46.63% from suggestions). Note that SHIPWRIGHT used the patterns produced by analyzing clustered files. That was possible because the clustering step is conservative and clusters were based on embeddings of largely syntactic information (logs). For example, we observed that a file failing on the statement `apk add A && apk add B && ... && apk add bzr` was not clustered with other files failing on `apk add bzr`—but, upon further examination, we found that this file failed to cluster due to its use of a conjunction of successive `apk add` commands instead of the (more common) use of the multi-argument `apk add A B ... bzr` variant. In practice, although conservative, the generated clusters were suitable for creating useful and generalizable repairs.

Finally, even when no repairs or suggestions apply, SHIPWRIGHT can still provide a list of URLs pointing to resources that may provide a developer with a fix for their broken file.

Summary of RQ3.3: Even in non-clustered broken Dockerfiles, SHIPWRIGHT was able to produce automated repairs in 18.18% of the files. Furthermore, when no repairs applied, SHIPWRIGHT was able to provide suggestions in 46.63% of the files.

D. Answering RQ4: How effective is SHIPWRIGHT *in reducing the number of broken Dockerfiles in public repositories?*

This section reports on two experiments we conducted to assess the practical usefulness of SHIPWRIGHT. The first experiment (Section V-D1) measures the fraction of initially-broken but later-fixed Dockerfiles that could have been repaired with SHIPWRIGHT. The second experiment (Section V-D2) measures the acceptance ratio of Pull Requests (PRs) for Dockerfiles found to be still broken in their repositories.

1) Repair Confirmation: This experiment evaluates SHIPWRIGHT on real patches created by GitHub developers. The metric we used was the fraction of the patches created by developers that matched the repairs or suggestions of SHIPWRIGHT. To run this experiment, we searched for fixed Dockerfiles on GitHub. We used the same procedure as reported in Section IV-A, but we re-cloned the repositories on Aug. 14, 2020 (8/14/20). Because we know that the Dockerfile build on the first version of the project failed, we only needed to perform Dockerfile builds for the 8/14/20 versions of projects. To avoid unnecessary builds, we looked for Dockerfiles that were changed in the repository, and found that 161 (=8.87%) of the original 1,814 broken Dockerfiles were changed in their repositories from the day they were retrieved up to 8/14/20. We ran the command `docker build` in-context on those 161 files, and discarded the cases where the build was still unsuccessful. In the end, we obtained a set of 102 $\langle x, y \rangle$ pairs to analyze, with x denoting a broken Dockerfile from our dataset and y denoting its corresponding patch. The method we used to measure effectiveness of SHIPWRIGHT was to run SHIPWRIGHT on x and compare the generated repair or suggestion, if found, with y.

Of the 102 cases of initially-broken then-fixed Dockerfiles, SHIPWRIGHT produced an identical repair in 23 of the cases. In 77 cases, SHIPWRIGHT provided suggestions that matched the patch used by the developer. Although we found that the ratio of suggestions to fixes was higher compared to results of RQ3.1, SHIPWRIGHT covered most of the cases we analyzed (a total of 98.04% of the cases). Overall, we believe that this result is encouraging because it provides a strong (and relatively unbiased) indication that the repairs that SHIPWRIGHT produces are (i) correct (they matched the fixes of developers) and (ii) useful (almost all cases were covered).

2) Pull Requests (PRs): This experiment evaluates SHIPWRIGHT on Pull Requests (PRs) issued to GitHub projects with still-broken Dockerfiles. The goal is to assess the feedback from developers to these PRs, which is a proxy for their interest in SHIPWRIGHT's results. For each of the 13 repair patterns, we randomly sampled 5 Dockerfiles (from our dataset) that remained broken until the date we ran this experiment. Then, we manually prepared a PR that explained the problem (including a link to a similar case) and proposed a repair, as created by SHIPWRIGHT. To avoid violating double-blind rules, we created and used a GitHub account under the fictitious name "Joseph Pett" to submit the PRs. Our artifact (https://github.com/STAR-RG/shipwright) includes an up-to-date tracker of the submitted, accepted, and rejected PRs.

Of the 45 PRs that we submitted, 19 were accepted by developers (=42.2%); 4 PRs were rejected; and 22 PRs have not yet been reviewed by developers. The number of submitted PRs was lower than 65 (=13*5) because we could not find five Dockerfiles still broken for some of the patterns.

Three of the four rejected PRs were related to the same organization and the same problem, characterized by pattern #7 (Table I). The developer pointed out that using a new version of the Docker Ruby image solved the encoding problem, and he preferred to update the Ruby version. With that feedback, we revised repair #7 to include a second solution, which is to update the Ruby version to 2.5.8. We have confirmed that this repair also works for the Dockerfiles repaired by the original solution. The new version of the Ruby image was committed on June 2020 [31], while this issue has been reported since June 2015 [25]. This GitHub issue was the URL recommended by SHIPWRIGHT to assist the human to produce a repair.

Summary of RQ4: These results provide initial, yet strong, evidence that SHIPWRIGHT is a useful aid to help developers fix broken Dockerfiles.

VI. THREATS TO VALIDITY

Although most of our analysis is based on samples of (broken) Dockerfiles from GitHub repositories with ten or more stars, it is possible that this data is not representative of Dockerfile use in general. Nonetheless, we found that real developers accepted the patches that SHIPWRIGHT generated, and thus we can be reasonably sure that the trends (and repairs) we have identified are applicable in practice. Additionally, our repairs and suggestions require a human in the loop, which is a source of bias. To side-step this source of bias, we ran a "time-travel" analysis in which we were able to confirm retroactively that SHIPWRIGHT's repairs and suggestions were either identical (for repairs) or similar (for suggestions) to patches that developers actually applied. This study is an important counterpoint to our pull-request study, because even pull-request acceptances could be biased, in the sense that it is hard for a developer to "reject" an offered patch.

We also made efforts to bolster our results by using robust methodology where possible: e.g., to understand clustering behavior, we ran a grid search over 200 different configurations of hyper-parameters; we also benchmarked two recent static tools to give some context to SHIPWRIGHT results.

VII. RELATED WORK

Empirical studies on Docker (and DevOps). A growing number of studies have been carried out on Dockerfiles, as well as on the broader topic of DevOps [32] (also known as *infrastructure as code*). For Docker, Cito *et al.* [33] examined Dockerfile quality and, similar to us, found a high rate of breakage in Dockerfile builds; they cite a 34% breakage rate from a smaller sample of 560 projects. We found a comparable breakage rate, but have also developed methods aimed at making *repairs* instead of just analyzing quality. More recently, Wu *et al.* [34] conducted a comprehensive study of build failures in Dockerfiles. They analyzed a total of 3,828 GitHub project containing Dockerfiles, and a total of 857,086 Docker builds. Overall, they found a failure rate of 17.8%. Despite the differences in failures rates, these studies corroborate our finding that build failures are prevalent. Lin *et al.* [35] analyzed patterns (i.e., good and bad practices) in Dockerfiles. Among various observations, they found that many Dockerfiles use obsolete OS images, which can pose security risks (because

attackers could exploit documented vulnerabilities) and incorrectly use the latest tag. Xu and Marinov [36] investigated characteristics of Docker images from DockerHub. Among other findings, they listed opportunities to improve Software Engineering tasks based on how images are organized. For example, they report that image variants could be used to support combinatorial testing. Zerouali *et al.* [37] studied version-related vulnerabilities (yet another category of issues that may arise in Dockerfiles—similar to some of the build-breaking issues we observed, in which *external changes in the environment* negatively effect a Dockerfile). Among various findings, they found that no release is devoid of vulnerabilities, so deployers of Docker containers cannot avoid vulnerabilities even if they deploy the most recent packages.

Analysis of Dockerfiles Henkel *et al.* [29] created a static checker for Dockerfiles (similar to Hadolint [2]), called `binnacle`, which is capable of learning rules from existing Dockerfiles; however, unlike SHIPWRIGHT, neither `binnacle` nor Hadolint attempts *repairs*. Xu *et al.* [38] examined "Temporary File Smells", which are an *image-quality*-related issue, not a *build-breaking* issue, such as the ones we examined. Zhang *et al.* [39] studied the effect of Dockerfile changes on build time and quality (and utilized the static tool Hadolint). Hassan *et al.* [40] proposed RUDSEA a tool-supported technique that proposes updates in Dockerfiles. RUDSEA analyzes changes in software environment assumptions—obtained with static analysis—and their impacts. We consider RUDSEA and SHIPWRIGHT to be complimentary approaches: RUDSEA focuses on changes within a project and SHIPWRIGHT focuses on changes external to a project. Other empirical studies on DevOps, but not Docker, include an examination of *smells* in software-configuration files [41], and a study of the coupling between infrastructure-as-code files and "traditional" source-code files [42].

Automated Code Repairs. SHIPWRIGHT lies within the growing body of work in automated repair. According to a recent survey [43], our approach can be classified as both *Generate-and-Validate* and *Fix Recommender*. We use predefined templates that are obtained (i) via the analysis of build logs extracted from our clusters, and (ii) from examples found in community websites. As such, we side-step the challenge of a fully automatic repair process to produce acceptable fixes. In addition to automated repair of source code, there is a growing effort to automate repair of build-related (DevOps) artifacts. These DevOps artifacts are unique in that they are often tied to both a source repository and the broader external environment in which one wants to build, test, and/or run their code. In the broader context of repair for build-related artifacts, both Lou *et al.* [44] and Hassan and Wang [4] investigate repair in the domain of Gradle builds (a kind of DevOps artifact used in many Java projects) and Macho *et al.* [45] explore the related problem of automated repair for Maven builds.

Broken Updates in Package Managers. Prior work investigated the impact of breaking changes in package managers. Mancinelli *et al.* [14] formalized package dependencies within a repository, and encoded the installability problem as a SAT problem. Vouillon and Cosmo [10] proposed an algorithm to identify *broken sets* of packages that cannot be upgraded together within a component repository. McCamant and Ernst [11] proposed an approach for checking incompatibility of upgraded software components. They compute operational abstractions based on input/output behaviour to test whether a new component can replace an old one. Møller and Torp [12] proposed a model-based testing approach to identify type-regression problems that result in breaking changes in JavaScript libraries. These works deal with improvements and repairs applied to a package repository or library, and thus have a different focus compared to SHIPWRIGHT, which is on repairing *broken* Dockerfiles. More related to checking inconsistenciesof client code, Tucker *et al.* [13] proposed the OPIUM package-management tool. Given a set of installed packages and information about dependencies and conflicts, they used a variety of solvers to determine (i) if a new package can be installed; (ii) the optimal way to install it; and (iii) the minimal number of packages (possibly none) that must be removed from the system. SHIPWRIGHT does not rely on explicit information about dependencies (which might not be available or feasible to obtain). Instead, it extracts information from build logs, and leverages community knowledge bases to find solutions. This approach enables SHIPWRIGHT to address problems that go beyond broken packages and conflicts.

VIII. CONCLUSIONS

In an analysis of many open source repositories that use Docker, we found a surprising number of *broken* Dockerfiles, most of which existing static analyzers cannot detect. For the cases they can detect, they do not propose repairs. To address this problem, we propose SHIPWRIGHT, a human-in-the-loop approach for clustering, analyzing, and fixing broken Dockerfiles. We conducted a comprehensive evaluation of SHIPWRIGHT, which showed that it was a helpful aid to fix broken Dockerfiles on GitHub. Using SHIPWRIGHT we were able to submit 45 pull requests, of which 19 were accepted. Furthermore, the tools and data we produced as result of this study are publicly available online via our artifact:

https://github.com/STAR-RG/shipwright

.

ACKNOWLEDGMENTS

Supported, in part, by a gift from Rajiv and Ritu Batra; by Facebook under a Probability and Programming Research Award; and by ONR under grants N00014-17-1-2889 and N00014-19-1-2318; by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, FACEPE grant APQ-0570-1.03/14 and CNPq (grants 465614/2014-0, 309032/2019-9, 406308/2016-0, 409335/2016-9). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

REFERENCES

[1] Portworx, "Annual Container Adoption Report," Apr 2017, [Online; accessed 21. Aug. 2019]. [Online]. Available: https://portworx.com/2017-container-adoption-survey/

[2] "hadolint/hadolint," Aug 2019, [Online; accessed 21. Aug. 2019]. [Online]. Available: https://github.com/hadolint/hadolint

[3] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017, e1838 smr.1838. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1838

[4] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1078–1089. [Online]. Available: https://doi.org/10.1145/3180155.3180181

[5] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "A dataset of dockerfiles," 2020.

[6] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. [Online]. Available: http://arxiv.org/abs/1908.10084

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *Proceedings of the 2019 Conference of the North*, 2019. [Online]. Available: http://dx.doi.org/10.18653/v1/N19-1423

[8] R. J. G. B. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *Advances in Knowledge Discovery and Data Mining*, J. Pei, V. S. Tseng, L. Cao, H. Motoda, and G. Xu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 160–172.

[9] docker, "Docker hub: Database of container images," 2015, hub.docker.com.

[10] J. Vouillon and R. D. Cosmo, "Broken sets in software repository evolution," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 412–421.

[11] S. McCamant and M. D. Ernst, "Early identification of incompatibilities in multi-component upgrades," in *ECOOP 2004 – Object-Oriented Programming*, M. Odersky, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 440–464.

[12] A. Møller and M. T. Torp, "Model-based testing of breaking changes in node.js libraries," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 409–419. [Online]. Available: https://doi.org/10.1145/3338906.3338940

[13] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, "Opium: Optimal package install/uninstall manager," in *29th International Conference on Software Engineering (ICSE'07)*, May 2007, pp. 178–188.

[14] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the complexity of large free and open source package-based software distributions," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, Sept 2006, pp. 199–208.

[15] A. Barcelona, "Dependendy on ruby version 2.6.5," 2020, https://github.com/AjuntamentdeBarcelona/decidim-barcelona/blob/83ef28ee6af9d7ec2ac7914762c00db165592615/Gemfile#L5.

[16] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96. AAAI Press, 1996, p. 226–231.

[17] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[18] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[19] oshivwanshi, "RROR: Error installing bundler: bundler requires ruby version," 2019, https://github.com/rubygems/bundler/issues/6865.

[20] r1williams, "Rpmdb checksum is invalid: dcdpt(pkg checksums)," 2015, https://github.com/CentOS/sig-cloud-instance-images/issues/15.

[21] D. Schulze, "apt-get update fails on 17.04 [closed]," 2018, https://askubuntu.com/questions/1059898/apt-get-update-fails-on-17-04.

[22] jahanzaib basharat, "E: Package 'libpng12-dev' has no installation candidate," 2018, https://github.com/docker-library/php/issues/662.

[23] PacificNW_Lover, "Install python-pip using apt-get via ubuntu's apt-get in dockerfile," 2020, https://stackoverflow.com/a/61564831.

[24] Tan, "How to fix your ruby version is 2.3.0, but your gemfile specified 2.2.5 while server starting," 2016, https://stackoverflow.com/questions/37914702.

[25] ubergesundheit, "Change locale to c.utf-8," 2015, https://github.com/docker-library/ruby/issues/45.

[26] rmNyro, "npm not found on latest build," 2017, https://github.com/gliderlabs/docker-alpine/issues/327.

[27] yelizariev, "bzr is not available in apline:," 2020, https://github.com/alpinelinux/docker-alpine/issues/87.

[28] Buddhi, "How to download a file using curl," 2019, https://stackoverflow.com/a/54735579.

[29] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "Learning from, understanding, and supporting devops artifacts for docker," 2020.

[30] Linnovate, "Mean stack," 2020, https://github.com/linnovate/mean.

[31] mtsmfm, "Set lang by default," 2020, https://github.com/docker-library/ruby/commit/8813cdda206acb36ea7797919bf8dadb84fc5ac7.

[32] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information & Software Technology*, vol. 108, pp. 65–77, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.12.004

[33] 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)*, May 2017, pp. 323–333.

[34] Y. Wu, Y. Zhang, T. Wang, and H. Wang, "An empirical study of build failures in the docker context," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 76–80. [Online]. Available: https://doi.org/10.1145/3379597.3387483

[35] C. Lin, S. Nadi, and H. Khazaei, "A large-scale data set and an empirical study of docker images hosted on docker hub," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 371–381.

[36] T. Xu and D. Marinov, "Mining container image repositories for software configuration and beyond," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, May 2018, pp. 49–52.

[37] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the relation between outdated docker containers, severity vulnerabilities, and bugs," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2019, pp. 491–501.

[38] 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)*, vol. 1, July 2019, pp. 185–190.

[39] 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. IEEE, 2018, pp. 138–143.

[40] 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*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 796–801. [Online]. Available: https://doi.org/10.1145/3238147.3240470

[41] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 189–200.

[42] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code-an empirical study," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 45–55.

[43] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[44] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang, "History-driven build failure fixing: How far are we?" in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 43–54. [Online]. Available: https://doi.org/10.1145/3293882.3330578

[45] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 106–117.