Assessing and Improving the Quality of Docker Artifacts

Giovanni Rosa
STAKE Lab
University of Molise
Pesche, Italy
giovanni.rosa@unimol.it

Simone Scalabrino
STAKE Lab
University of Molise
Pesche, Italy
simone.scalabrino@unimol.it

Rocco Oliveto
STAKE Lab
University of Molise
Pesche, Italy
rocco.oliveto@unimol.it

***Abstract*—Docker is the most diffused containerization technology adopted in the DevOps workflow. Docker allows shipping applications in Docker images, along with their dependencies and execution environment. A Docker image is created using a configuration file called Dockerfile. The literature shows that quality issues, such as violations of best practices (*i.e.,* Dockerfile smells), are diffused among Docker artifacts. Smells can negatively impact the reliability, leading to building failures, poor performance, and security issues. In addition, it is unclear to what extent developers are aware of those quality issues and what quality aspects are correlated with the adoption of a Docker image. As evaluated in the literature, composing high-quality Dockerfiles and Docker images is not a trivial task. In this research, we aim to propose approaches and techniques to assess and improve the quality of Dockerfiles and Docker images. First, starting from the resolution of Dockerfile smells, we aim to improve the internal and then the related external quality aspects that also affect the developers' preference and the perceived quality when they adopt a Docker image. Next, we want to employ that knowledge in the automated generation of high-quality Dockerfiles and Docker images.**

***Index Terms*—Docker, empirical software engineering, software evolution, software quality**

I. Problem and Research Hypothesis

DevOps is a methodology commonly adopted in modern companies: Development and operation teams work together to provide a quick and automated release cycle of software systems [1]. Containerization technologies are essential in the development workflow since they simplify the deployment of applications and provide an isolated runtime environment, along with all the dependencies and configurations required for the execution.

Docker[1] is one of the most popular platforms used in the DevOps workflow[2]. Since its release in 2013, Docker has become the most talked-about containerization technology, becoming the #1 "Most Loved" and "Most Wanted" platform in the 2022 StackOverflow survey [2]. A recent report on *cloud workload platforms* states that 85% of the organizations will adopt containerization in production environments by 2025, starting from less than 30% observed in 2020 [3].

Docker allows shipping applications with the required execution environment through lightweight virtual environments (*i.e.,* containers). A *Dockerfile* is a text file containing the configuration for the execution environment required by a specific application that generates a Docker image through a build process. When writing Dockerfiles, existing Docker images can be used as *base images*, *i.e.,* a base environment on which a new Docker image is built. To this end, a public online repository called DockerHub[3] allows developers to share their images, so that others can build upon them.

DockerHub provides several sets of functionally-equivalent environments (*i.e.,* which provide the same software packages). For example, if developers need an environment based on Apache Tomcat, they can choose as base image either *tomcat* or *bitnami/tomcat*. Developers might find it nontrivial to select the best base image for their application [4] [5] since they not always understand the *nonfunctional* implications of choosing a given image. The nonfunctional aspects can be divided in *externally observable* features and *configuration* features. The *externally observable* features are those that the image user sees, such as the image size. On the other hand, the *configuration* features are those aspects only perceivable by the developers, mainly related to Dockerfiles and the build process of the Docker image, such as the presence of Dockerfile smells. Similar to code smells that occur in software systems [6], Dockerfile smells [7] can negatively impact the overall quality of the resulting Docker image.

While there are static analysis tools that help developers verify the adherence to best practices in Dockerfiles, they may not be sufficient to assess the absence of code smells [8]. Several studies investigated the occurrence of those quality issues in Docker applications [9]–[12], where *hadolint* [13] is the reference tool to check for rule violations based on the official Docker guidelines. In the same direction, other studies evaluated the occurrence of issues affecting the security of Docker images [14]–[16]. Besides, a previous research reports that developers perceive the creation of Dockerfiles as a time-consuming activity [17]. In summary, there is a need for tools to support the development and release of those artifacts [7], [9], [11], [18]–[20].

To the best of our knowledge, no other studies perform a comprehensive evaluation of the aforementioned *externally observable* and *configuration* features in terms of (i) their

1. https://www.docker.com/
2. https://portworx.com/blog/2017-container-adoption-survey/
3. https://hub.docker.com/

relationship and (ii) the relationship with the developers' preferences when choosing a Docker image. Moreover, there are no previous studies that propose fixing approaches and tools in that direction, although developers perform refactoring operations to improve Dockerfiles [21]. Some approaches provide fixing recommendations, but for the automated repair of broken Dockerfiles [22] and fixing security vulnerabilities[4]. The lack of techniques to support the quality of Docker images and Dockerfiles, and consequently the refactoring of smells, is still an open challenge. Therefore, we want to focus our research activity on the definition of new approaches and tools to support the development and maintenance of Dockerfiles and Docker images. Specifically, we want to investigate the following *research directions*:

- **Identification and improvement of internal quality factors of Dockerfiles.** We want to improve the quality of Dockerfiles, proposing approaches to fix Dockerfile smells. Despite there are studies evaluating those issues, it is not clear which of them are relevant for developers. Thus, we want to identify what Dockerfile smells are relevant for developers and introduce approaches to fix them.
- **Identification of external (and internal) features that are correlated with the quality perceived by developers for Docker images.** We want to understand what Dockerfile- and Docker image-related features explain developers' preferences. Specifically, we want to understand (i) what externally observable features explain the developers' adoption and perceived quality, and (ii) what configuration features influence the externally observable features.
- **Quality-aware generation of Dockerfiles that produce Docker images intercepting users' preferences.** We want to support the automated generation of Dockerfiles starting from a natural language specification of the requirements. Such an approach has to take into account also the quality that the resulting Docker images will obtain. This means that the generated Dockerfile, when built, also produces a high-quality Docker image that developers are most likely to adopt.

II. Improvement of the Quality of Dockerfiles

Motivation. Several studies show that Dockerfile smells, are widely diffused among open-source Dockerfiles [7], [9], [11], [19]. *Hadolint* [13] is the reference tool for detecting Dockerfile smells that are commonly used in the literature as a proxy for Dockerfile quality. *Hadolint* checks for a set of rules on the AST representation of the input Dockerfile. The rules are directly derived from the official writing guidelines [23]. Each rule is identified by a unique identifier, composed of a prefix (*i.e., DL* and *SC*, for Docker instruction and shell script, respectively) and a number. An example is the rule *DL3020*[5], which checks the usage of the instruction `COPY` instead of `ADD` when copying files and folders. While the prevalence of each smell is known [19], it is still unclear what are the Dockerfile smells considered relevant by the developers, and to what extent they are interested in solving them. Fixing Dockerfile smells could reduce the risk of build failures [9], security issues [16], [24], and lower both build latency and the size of the resulting image [25]. To fill this gap, we want to conduct an empirical study to (i) investigate the survivability of code smells, to understand what are the most relevant and how developers fix them, and (ii) evaluate to what extent developers are willing to accept fix recommendations for those smells.

Methodology. The experimental procedure of our study is described as follows. As a study context, we plan to use the dataset built by Eng *et al.* [19], which is the largest ($\sim$9.4M instances) and most recent (*i.e.,* up to 2020) dataset of Dockerfiles available in the literature. To evaluate the smell survivability, we will run the *hadolint* tool on different snapshots of the same Dockerfile over time, to identify best practices violations (*i.e.,* smells). Considering the default configuration of the *hadolint* tool, we will only consider rule violations with a severity level of *error* and *warning*. Moreover, through a manual evaluation, we will aim to understand if and how developers performed the fixes. In this way, we can also exclude false positives (*e.g.,* the smell disappeared because the commit removed the smelly lines). We will also look for custom configurations of *hadolint* in the evaluated repositories, to understand which smells are *explicitly* considered relevant by developers.

For the second part of our study, we plan to implement a tool to recommend fixes for Dockerfile smells, based on the fixing examples included in the *hadolint* catalog[6]. The tool will include fixes for the most occurring smells (*i.e.,* those reported by Eng *et al.* [19]), excluding those that are equivalent but less occurring (*i.e.,* we retain *DL3008*, discarding *DL3013* and *DL3018*). While most of the smells are trivial to fix, those related to missing version pinning are more difficult and require a heuristic approach to be fixed. In detail, *DL3006* checks for the presence of the version tag for base images in Dockerfiles, and *DL3008* checks for the missing version pinning for software packages installed via `apt-get`. We plan to submit pull requests on GitHub to evaluate the fix recommendations. We will monitor the status of each pull request for 3 months and interact with developers for additional information. Also, we will perform a quantitative analysis of the status of the pull requests, and a qualitative analysis of the developers' reactions.

Expected outcome. The outcome will benefit both developers and researchers: Understanding what are the relevant smells for developers will provide insights on where to focus our research effort to improve the internal quality of Docker artifacts.

Preliminary achievements. We have submitted a registered report paper in which present the empirical study design. We have already implemented a modular tool for fixing smells, with the fixing procedures for some of the most common

4. https://snyk.io/docker/
5. https://github.com/hadolint/hadolint/wiki/DL3020
6. https://github.com/hadolint/hadolint/wiki

smells, namely *DL3006* (image version tag), *DL3008* (*apt-get* version pinning), *DL3009* (delete *apt-get* lists after installing packages), *DL3020* (use `COPY` instead of `ADD` for files and folders), *DL4000* (replace `MAINTAINER` with `LABEL`), and *DL4006* (not using `-o pipefail` before `RUN`). Finally, we verified if the fixed Dockerfile builds correctly to assess the correctness of those fixes.

Limitations. Our study heavily depends on the *hadolint* tool. The limitations of such a tool in detecting smells directly impact our study. Another risk is that fixing Dockerfile smells could not be a priority for developers and, thus, we could find that no smell is relevant to them. However, as shown in previous studies, there is a decreasing trend in the occurrence of some smells [9], [11], [19]. This suggests that developers probably fix some smells.

III. Identification of Feature Correlated with the Developers' Preference for Docker Images

Motivation. There are many Docker images publicly available on DockerHub that can be used by developers, but the selection of the correct image is a nontrivial task because of the countless alternatives providing the same software applications [5]. However, from Docker artifacts, it is possible to extract *configuration* features, related to the Dockerfile and the build process of the image (*e.g.,* Dockerfile smells), and *externally observable* features, related to the Docker image (*e.g.,* image size) that developers and Docker image users can observe. Previous work proposed metrics to evaluate Docker images through some of those features [7], [9], [11], [19], [21], [25]. However, it is still unclear why developers prefer a Docker image over another in terms of those features, and how the configuration features are correlated with the externally observable ones, thus affecting adoption and perceived quality. The aim of our research work is to fill this gap. We want to perform a comprehensive evaluation of the configuration features and external features of Docker images including their relation. The aim is to evaluate how externally observable features are correlated with developers' preferences and the quality perception of Docker images, and what configuration features are correlated with those external features.

Methodology. The goal of the empirical evaluation that we want to conduct is two-fold. On one hand, we investigate the scientific literature looking for all the features related to quality, along with the metrics proposed to measure them. The output of this process is a taxonomy of those features. On the other hand, we want to empirically evaluate the correlation of that features with the developers' preference when adopting a Docker image (*i.e.,* how frequent they appear as base images in Dockerfiles from open-source repositories) and the quality that they perceive (*i.e.,* how much they "appreciate" a Docker image, measured as *stargazers* count from DockerHub). Most of these features can be automatically extracted from Dockerfiles and Docker images, if provided by DockerHub. Those that require a manual effort are excluded from the evaluation (*e.g.,* temporary file smell [8]).

Expected outcome. The expected outcome will be (i) a taxonomy of externally observable and configuration features, and (ii) several models that explain how configuration features determine each externally observable feature and how externally observable features determine the perceived quality. Researchers will obtain a reliable set of metrics to estimate the quality of Docker artifacts, instead of measuring only the number of smells, that also takes into account the quality perceived by developers. Tool builders will know where to put effort when they want to improve the quality of Docker images and Dockerfiles. As the next steps, a quantitative measure of Docker image and Dockerfile quality can be proposed by combining configuration and external features. The combination of their metrics can allow providing an overall score of the quality level (*i.e., quality badge*). Such a measure can be used in refactoring tools for Dockerfiles to verify if the quality of the resulting Docker image is improved. Also, the *quality badge* can be used as a discriminant factor (*i.e.,* qualitative measure) in recommending systems, or simply it can be integrated with DockerHub as a meta-information to show for Docker images.

Preliminary achievements. We conducted the whole empirical evaluation. In detail, our empirical study answers two different research questions, namely (i) how the externally observable features are correlated with the developers' preference for a Docker image, and (ii) how the configuration features are correlated with the external ones. With the first one, we find out the externally observable features correlated with the users' choice when adopting an image. With the second one, we identify what configuration features influence those externally observable that are more relevant in terms of correlation. In this way, we evaluate the configuration features that are also indirectly correlated with the developers' preferences. The results show that developers are more likely to adopt Docker images labeled as "official" on Docker Hub. They are also better perceived, as shown in a previous study [5]. The image size and the presence of exposed secrets (*e.g.,* exposed login data) have a negative impact on developers' preferences. Also, the number of security vulnerabilities negatively impacts the perceived quality. In terms of configuration features, the SLOC, measured on Dockerfiles, have a positive and indirect impact on developers' preference because it directly impacts, negatively, the number of vulnerabilities and the image size.

Limitations. As we rely on the scientific literature on Docker that is relatively recent, there can be quality features not identified yet. In the future, we plan to conduct a developer survey to (i) validate the features considered in the presented study and (ii) find out if there are additional features that are relevant but not evaluated.

IV. Quality-aware Generation of Dockerfiles

Motivation. Creating a Dockerfile is not trivial, because of the selection of the correct base image [4], [5], or the knowledge required to resolve all the software dependencies [26]. In previous studies, researchers highlighted the need for tools that support the creation of Dockerfiles as developers considered it a time-consuming activity [17]. Previous studies made the first steps in this direction. Hanamaya *et al.* [27] proposed Humpback, a tool that provides code completion for Dockerfiles via language models, created using LSTM neural networks. Horton *et al.* [28] proposed *DockerizeMe*, an automated technique to generate entire Dockerfiles for Python projects using a graph-based inference procedure. Ye *et al.* [26] proposed *DockerGen*, an approach that automatically containerizes software packages, generating a Dockerfile specification by simply providing the target OS (optional) and the target software. DockerGen relies on knowledge graphs which contains associations between packages, built on a dataset of 220k Dockerfiles, to generate the `FROM` instruction and the `RUN` instructions with package installation commands. While DockerGen is a milestone, it is still limited since it only support package installation instructions, while Dockerfiles might contain more complex instructions to install external dependencies or to initialize the environment. Therefore, our objective is to introduce a novel approach that automatically generates Dockerfiles using Deep Learning and, specifically, the T5 model, which has been proven effective in other code-related tasks [29].

Methodology. Our approach takes as input a natural language specification of the desired environment. Then, the T5 model takes as input the specification and generates a Dockerfile. To train such a model it is necessary to build a large dataset of Dockerfiles associated with the respective specification. We will rely on the largest dataset proposed in literature (*i.e.,* Eng *et al.* [12]), which contains $\sim$9M Dockerfiles. Given a Dockerfile, one of the biggest challenge will be to infer the specification behind it so that we can train the model. As shown in previous studies, developers usually provide comment lines in Dockerfiles to explain what each instruction does in Dockerfiles [20]. Therefore, to infer the specification based on which a Dockerfile was written (*e.g.,* operating system, software packages, and package manager), we will define a parser that extracts such information from code comments along with source code. Consider, for example, the code comment `# Install Python`, followed by a `RUN` instruction that installs `python2.7` and `python-pip`: The comment clearly indicates that *python* is the abstract software requirement needed by the developer, while `python2.7` and `python-pip` are the actual packages that the approach must generate.

To validate our approach, we plan evaluate the generated Dockerfiles in terms of (i) adherence to the specification, (ii) similarity between the generated Dockerfile and the actual Dockerfile from which the specification has been extracted, and (iii) the functional equivalence of the resulting Docker image. For the first point, we plan to measure the similarity between the input specification and the one created from the generated Dockerfile. For the second point, we will evaluate the edit distance between the ASTs of the generated Dockerfile and the source one used to parse the input specification. For the last point, we will check whether the generated Dockerfile can be successfully built and whether the composition similarity, using the hash of the layers, between the resulting Docker image and the one built from the source Dockerfile.

Expected outcome. We aim to propose an approach that generates the entire Dockerfile starting from a natural language specification, where developers indicate the requirements to satisfy. The resulting tool can be integrated into the IDE, providing the generated Dockerfile as a recommendation. In that case, more than one capable Dockerfile could be generated. A discriminant factor for the generated Dockerfiles is the overall quality of the code, which can be measured in terms of number of smells [9], or even better, considering the features that influence the developers' preference and the quality perceived (see Section III).

Limitations. The extraction of software requirements strongly relies on the presence of comments in Dockerfiles. Due to the size of the source dataset, removing Dockerfile that does not contain comments is not an issue. Also, it is possible that developers do not explicitly indicate some of the requirements. Therefore, it is possible that the generated Dockerfiles contain only a subset of the instructions of the Dockerfiles used for training them.

V. Final Remarks

The presence of quality issues related to Docker images and Dockerfiles is a common problem for developers. Despite scientific research on source code quality being widely investigated, the quality assessment of infrastructure as code (IaC) is still in its infancy. Our work will lay the groundwork for a research line that aims to provide those techniques to (i) support developers in defining Dockerfiles and (ii) improve the quality of Dockerfiles and Docker images. As a future research agenda, we plan to develop further our research.

First, following up our research on the importance of Dockerfile smells for developers, we plan to propose a comprehensive refactoring recommendation approach and tool for the most common and relevant Dockerfile smells. Second, as an extension of research on the impact of *externally observable* and *configuration* features on developers' preferences, we plan to define a quality metric for Docker images based on those features and to propose a *quality badge*. We plan to conduct a developer survey to empirically validate the reliability of that measure. Third, we will experiment with the use of different quality-based filters (*e.g.,* number of smells or the previously-mentioned quality metric) to select the instances to use to train our approach for the generation of Dockerfiles to understand if it allows to achieve better results.

We believe that the results of our research will produce concrete support for software developers, and give insights on where to put the effort in successive research studies on this topic.

References

[1] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "Dimensions of devops," in *International conference on agile software development*. Springer, 2015, pp. 212–217.

[2] StackOveflow, “2022 developer survey,” https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-other-tools, [Online; accessed 7-Jul-2022].

[3] Gartner, “2021 gartner market guide for cloud workload protection platforms,” https://businessresources.bitdefender.com/gartner-2021-market-guide-for-cloud-workload-protection-platforms?hs_preview=CPRimYYO-51790146713&hsLang=en-us, [Online; accessed 27-Jun-2022].

[4] A. Brogi, D. Neri, and J. Soldani, “Dockerfinder: multi-attribute search of docker images,” in *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2017, pp. 273–278.

[5] M. H. Ibrahim, M. Sayagh, and A. E. Hassan, “Too many images on dockerhub! how different are images for the same system?” *Empirical Software Engineering*, vol. 25, no. 5, pp. 4250–4281, 2020.

[6] P. Becker, M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[7] Y. Wu, Y. Zhang, T. Wang, and H. Wang, “Characterizing the occurrence of dockerfile smells in open-source software: An empirical study,” *IEEE Access*, vol. 8, pp. 34 127–34 139, 2020.

[8] 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. 63 650–63 659, 2019.

[9] 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)*. IEEE, 2017, pp. 323–333.

[10] 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*, 2020, pp. 76–80.

[11] 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)*. IEEE, 2020, pp. 371–381.

[12] K. Eng and A. Hindle, “Replication package of ”revisiting dockerfiles in open source software over time”,” Jan 2021.

[13] “hadolint: Dockerfile linter, validate inline bash, written in haskell,” https://github.com/hadolint/hadolint, [Online; accessed 28-May-2022].

[14] R. Shu, X. Gu, and W. Enck, “A study of security vulnerabilities on docker hub,” in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 269–280.

[15] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, “Docker ecosystem–vulnerability analysis,” *Computer Communications*, vol. 122, pp. 30–43, 2018.

[16] 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)*. IEEE, 2019, pp. 491–501.

[17] D. Reis, B. Piedade, F. F. Correia, J. P. Dias, and A. Aguiar, “Developing docker and docker-compose specifications: A developers’ survey,” *IEEE Access*, vol. 10, pp. 2318–2329, 2021.

[18] 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)*. IEEE, 2020, pp. 38–49.

[19] K. Eng and A. Hindle, “Revisiting dockerfiles in open source software over time,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 449–459.

[20] H. Azuma, S. Matsumoto, Y. Kamei, and S. Kusumoto, “An empirical study on self-admitted technical debt in dockerfiles,” *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–26, 2022.

[21] E. Ksontini, M. Kessentini, T. d. N. Ferreira, and F. Hassan, “Refactorings and technical debt in docker projects: An empirical study,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 781–791.

[22] J. Henkel, D. Silva, L. Teixeira, M. d’Amorim, and T. Reps, “Shipwright: A human-in-the-loop system for dockerfile repair,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1148–1160.

[23] “Best practices for writing dockerfiles,” https://docs.docker.com/develop/develop-images/dockerfile_best-practices/, [Online; accessed 2-Jun-2022].

[24] 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)*. IEEE, 2020, pp. 38–49.

[25] 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.

[26] H. Ye, J. Zhou, W. Chen, J. Zhu, G. Wu, and J. Wei, “Dockergen: A knowledge graph based approach for software containerization,” in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 986–991.

[27] K. Hanayama, S. Matsumoto, and S. Kusumoto, “Humpback: Code completion system for dockerfiles based on language models,” in *Proc. Workshop on Natural Language Processing Advancements for Software Engineering*, 2020, pp. 1–4.

[28] 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)*. IEEE, 2019, pp. 328–338.

[29] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, “Studying the usage of text-to-text transfer transformer to support code-related tasks,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 336–347.