One Size Does Not Fit All: An Empirical Study of Containerized Continuous Deployment Workflows

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

Bogdan Vasilescu
Carnegie Mellon University, USA
vasilescu@cmu.edu

Huaimin Wang
National University of Defense Technology, China
hmwang@nudt.edu.cn

Vladimir Filkov
DECAL Lab, University of California, Davis, USA
filkov@cs.ucdavis.edu

ABSTRACT

Continuous deployment (CD) is a software development practice aimed at automating delivery and deployment of a software product, following any changes to its code. If properly implemented, CD together with other automation in the development process can bring numerous benefits, including higher control and flexibility over release schedules, lower risks, fewer defects, and easier on-boarding of new developers. Here we focus on the (r)evolution in CD workflows caused by *containerization*, the virtualization technology that enables packaging an application together with all its dependencies and execution environment in a light-weight, self-contained unit, of which Docker has become the de-facto industry standard. There are many available choices for containerized CD workflows, some more appropriate than others for a given project. Owing to cross-listing of GitHub projects on Docker Hub, in this paper we report on a mixed-methods study to shed light on developers' experiences and expectations with containerized CD workflows. Starting from a survey, we explore the motivations, specific workflows, needs, and barriers with containerized CD. We find two prominent workflows, based on the automated builds feature on Docker Hub or continuous integration services, with different trade-offs. We then propose hypotheses and test them in a large-scale quantitative study.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*;

KEYWORDS

Continuous Deployment, Containerization, Docker, GitHub

ACM Reference Format:
Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. 2018. One Size Does Not Fit All: An Empirical Study of Containerized Continuous Deployment Workflows. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236033

1 INTRODUCTION

Continuous deployment (CD), also referred to as continuous delivery,[1] is the fast-paced, automation-heavy software engineering approach in which teams work in short iterations to produce software that is deployable (production ready) at any time [31]. CD has promised to deliver a revolution over the twice-yearly or so standard for software releasing, through greater control and flexibility over feature releases, incremental deployment of value, lower risks, fewer defects, easier on-boarding of new developers, less off-hours work, and a considerable uptick in confidence [45]. It is not surprising then that the Perforce report [49] found that 65% of software developers, managers, and executives have used CD. Moreover, the "State of DevOps" survey [52], with 3,200 participants from around the world, found CD positively impacts IT performance and negatively impacts deployment pain.

On the other hand, industry reports and academic studies have found that implementing the automation, *i.e.*, pipelines (workflows), needed to properly provide CD is challenging and takes a lot of time and tuning, due to the many moving parts and the specific needs of each project or organization [8, 31, 49, 56]. A prototypical CD workflow involves a continuous integration (CI) service, like Jenkins [60] and Travis [41], which is triggered by new changes in the version control system to build, test, and deploy the packaged application. Many separate and often redundant tools can be pipelined to assemble a CD system for a project or organization. How-to guides exist [2, 30], and turn-key solutions, mainly commercial, are also available [34]. Previous studies have looked at implementations of CD in individual organizations [45, 55, 63] and one study has compared implementations in 15 different organizations [56]. It is commonly reported in those studies that the benefits gained are many but that implementing CD takes time. As more and more experience is being gained with different ways to implement CD, it has become obvious that different solutions, *i.e.*, workflows, are possible, and that they may fit different needs. Stated differently, choosing one available workflow vs another may make a big difference to a specific project. Thus, maps of project needs onto prototype CD workflows would be considered helpful.

Here we focus on studying the (r)evolution in CD workflows caused by *containerization*, the virtualization technology that enables packaging an application together with all its dependencies and execution environment in a light-weight, self-contained unit.

[1] Although technically continuous deployment encompasses continuous delivery, the two terms tend to be used interchangeably in practice by developers. We don't distinguish between the two here; we define the workflows precisely, below.

Containerization has transformed CD workflows, promising additional speedups and higher level of abstraction. Containers (or *images*) encapsulating a packaged application ready for deployment can be specified declaratively, versioned together with the rest of the infrastructure code, built automatically, and published to some cloud-based registry for easy access by users and other applications. Among containerization technologies, *Docker containers* have become the de-facto industry standard. Since inception in 2013, Docker containers have been downloaded 29B+ times[2] and their usage is spreading rapidly; *e.g.*, the "Annual Container Adoption" report [50] found that 79% of companies chose Docker as their primary container technology. Thus, studying Docker container usage is relevant to most of the containerization community.

In this paper we seek to aid in deciding how to appropriately choose between Docker-enabled CD workflows, by collating lessons learned and offering data-driven evidence from different CD implementations in open source software (OSS) projects. We focus on OSS projects on GitHub, the largest public code repository host, and Docker Hub[3], the most popular cloud-based registry for Docker containers, which hosts over 2 million Docker image repositories as of March 2018; 94% of these images are linked to a GitHub repository, enabling the data mining for our study.

As with any CD pipeline, developers have considerable freedom to define custom Docker-enabled workflows, choosing, *e.g.*, what CI service to use, what to include in the images, and how to automate their construction and publication. Starting from these two public sources of data, GitHub and Docker Hub, in this paper we report on a mixed-methods study to explore the following questions:

RQ1: *What motivations, unmet needs, and barriers do developers face with their Docker-enabled CD workflows?* (see §3)
RQ2: *What are the differential benefits among specific Docker-enabled CD workflows?* (see §4)

The first part of our study is a multi-stage 150+ developer survey, the results of which revealed several common CD implementations in projects publishing images to Docker Hub: while some projects write their own scripts to deploy images, most use available tools which fit together without extensive retooling with their existing solutions, *e.g.*, standard CI services like Jenkins, Travis, and CircleCI. We found that two Docker image deployment workflows[4] were most prominent:
(1) a *Docker Hub auto-builds Workflow* (denoted *DHW*), where the registry itself builds the image automatically whenever GitHub source files change; and
(2) a *CI-based Workflow* (denoted *CIW*), where CI tools build images during the build and test stage, then publish to Docker Hub.

Additionally, the survey answers also generated hypotheses related to specific CD workflow outcomes: release frequency, build results, stability, and build latency, such as *image build latency tends to worsen over time*, and *CIW tends to have higher image release frequency than DHW* (see §3.4 and §3.6). To test these hypotheses, in the second part of our study we performed data gathering and statistical modeling. We collected data from 1,125 projects on Docker Hub, measuring build latency, release frequency, config file sizes and changes, commit sizes, testing times, and discussion lengths. The above four outcomes (release frequency, build results, stability, and build latency) were regressed against an extensive set of variables, fitted to the processed data, and well fitting models were obtained. In summary, we found that:

- CIW is associated with higher image release frequency than DHW. But over time, the release frequency of both workflows tends to drop;
- Image build latency tends to increase over time. Interestingly, CIW tends to have shorter build latency than DHW;
- Image build configuration stability tends to increase over time. But CIW tends to have lower Dockerfile stability than DHW;
- CIW is associated with more image build errors than DHW;
- There are notable differences *within* CIWs, not just between the DH and CI workflows.

Our survey questions, scripts, and data are online at https://github.com/yangzhangs/cd_replication.

2 BACKGROUND AND RELATED WORK

Docker and Docker Hub. Docker (https://www.docker.com) is an OSS project implementing operating system-level virtualization; it builds on many technologies from operating systems research: LXC [22] (Linux Containers), virtualization of the OS [7], *etc.* The technology is primarily intended for developers to create and publish *containers* [39]. With containers, applications can share the same operating system and, whenever possible, libraries and binaries [6]. The content of the container is defined by declarations in the `Dockerfile` [40] which specifies the Docker commands and the order of their execution. Docker launches its containers from *Docker image*, which is a series of data layers on top of a base image [23]. When developers make changes to a container, instead of directly writing the changes to the image of the container, Docker adds an additional layer with the changes to the image [42]. Since production environment replicas can be easily made in local computers, developers can test their changes in a matter of seconds. Also, changes to the containers can be made rapidly as only needed sections are updated following a change. This makes Docker very suitable for CI and CD implementations [1].

Existing studies related to Docker containers have typically focused on performance aspects [43], security vulnerabilities [20], and basic usage [17]. In particular, Cito *et al.* [17] found that deployment pipelines are, in most part, structured in multiple and consecutive phases, but that, thus far, there has been little research on Docker-enabled workflows in CD processes. A recent study has pointed out that software engineering tasks can benefit from the mining of container image repositories, like Docker Hub [65].

Docker Hub is Docker's cloud-based registry, containing 2,018,057 Docker images as of March 2018. Docker Hub provides GitHub integration as well as some featured tools, *e.g.*, automated builds [29], which allow developers to build their images automatically from GitHub sources [7]. The build data and Dockerfile information on Docker Hub is available for mining, if the repositories are public.

CD and Deployment Pipelines. Continuous Deployment (CD) is a practice in which incremental software updates are tested, vetted, and deployed to production environments [54]. CD leverages the

[2]https://www.docker.com/company, as of March 2018.
[3]https://hub.docker.com/
[4]These resemble the GitHub *push* and *pull-based* models: the CI workflow "pushes" the Docker image, while the DH workflow "pulls" it.

CI [41] process and extends it to include the immediate deployment of software [18]. Humble *et al.* [32] reported on an early overview of CD practices and introduced several guidelines. Vassallo *et al.* [63] investigated CD practices at ING [62], focusing on their impact on the development process and management of technical debt. Savor *et al.* [56] reported on an empirical study conducted in two high-profile Internet companies; they found that the adoption of CD does not limit scalability in terms of productivity in an organization even if the system grows in size and complexity. In summary, prior work mostly focused on defining CD and describing particular implementations in a small sample of organizations or projects.

The emergence of CD also increases the importance of deployment pipelines [36]. A deployment pipeline should include explicit stages, *e.g.*, building and packaging, to transfer code from a source repository to the production environment [3]. In each stage, developers can choose different tools or services, which, in turn, will produce different CD workflows. It is becoming increasingly clear that one size does not fit all, with recent studies by Shahin *et al.* [58], Zhao *et al.* [66], or Widder *et al.* [64] showing that the choice of CI/CD tools and infrastructures is highly context dependent.

Despite the importance of containerization and Docker in industry, to the best of our knowledge, no existing research has investigated the barriers and needs developers face when using containerized CD workflow, or what trade-offs developers must make when choosing different CD workflows. With this paper, we attempt to address this literature gap, and provide insights into the Docker-enabled CD workflows in the OSS community.

3 DEVELOPER SURVEY

Our study starts with a qualitative exploration of developers’ experiences and expectations using Docker containers as part of CD workflows (**RQ1**), for which we conducted a survey. Our goals were: (1) to gain understanding of how people use containerized CD, focusing on what *motivations*, *barriers*, and *unmet needs* developers face with their Docker-enabled CD workflows; and (2) to generate hypotheses to be tested in a follow-up quantitative study.

3.1 Survey Methods

Survey design and participants. Since little is published about containerized CD, we designed the survey broadly, around use cases and pain points, and ran a pilot to refine the protocol. Questions were inspired mainly by SE literature on trade-offs in CI [27] and online discussions about CD and Docker containers.[5] Specifically, the survey included multiple choice and open-ended questions, organized in four parts: (1) motivations for doing CD; (2) current CD tools and workflows; (3) unmet needs; and (4) barriers and pain points. We piloted the survey before full deployment. To obtain developer contact information, we first mined all projects with Docker images hosted on Docker Hub, that had source code repositories on GitHub.[6] Using the GitHub API, we then identified those projects’ owners and their contacts, sampled 1,000, and sent them email invitations with a link to the online form. We only surveyed project owners as we felt they would be most familiar with the overall development of the project.

[5]From the Docker forum, https://forums.docker.com
[6]Following “Source Repository” links on Docker Store pages, https://store.docker.com

Respondents and analysis. Within 10 days, we received 168 responses, for a response rate of 16.8%, consistent with other software engineering online surveys [51]. Respondents indicated that their experience in OSS was 8.6 years on average (median: 7; range 1—30), while their CI/CD experience was 4.3 years on average (median: 3; range 1—20). Additionally, 32 participants replied to our email invitations to show their interest in this research and to provide valuable suggestions and feedback. Since no question was mandatory, the number of responses per question may vary; we report actual numbers below. For open-ended questions, we used open coding [21] in two phases. During a first round, we carefully read the content of each answer and marked its keywords or statements. Later, during a second round, we iteratively aggregated the descriptions and summarized the categories. One author was involved in coding, all authors in discussion and refinements.

3.2 Motivations for doing CD

We start by gauging developer’s motivations for doing CD in general (open ended, 140 answers), to evaluate how much our population shares CD insights derived in other contexts. We uncovered a spectrum of motivations for doing CD (Table 1). Next, we contextualize these uncovered motivation categories with prior work, and illustrate each with representative quotes.

[M1] **CD helps us deploy automatically instead of doing it manually.** We expected that automation is a major concern for developers, and the survey answers confirmed that. E.g., R79 remarked “*because the project will be built automatically, no need for me to build the image and push it to the server*”. This is consistent with Neely *et al.* [45], who pointed out that it is important to eliminate all manual steps from a build in order to extend CI with release and deployment automation.

[M2] **CD gives us smoother and easier deployments.** As per Fowler [25], CI was presented as a way to avoid painful integrations. CD goes one step further to automate a software release, as it makes sure the software is production-ready, which provides developers with easier deployments. E.g., R71 responded, “*Continuous integration to accelerate the development and continuous deployment to simplify the passes to production reducing the complexity*”.

[M3] **CD allows us keep our production reliable.** Benefield [4] reported that the deployment infrastructure, coupled with intensive automated testing and fast rollback mechanisms, improves release reliability and quality. With CD, the deployment process and scripts are tested repeatedly before deployment to production, which keeps the production more reliable. E.g., R26 remarked, [CD is] “*convenient and more reliable, less error prone as a manual deployment*”.

[M4] **CD makes releasing faster.** Chen [8] reported that CD allows delivering new software releases to customers more quickly. By leveraging assistance provided with CD, project teams can release once a week on average, or more frequently if desired. Some of our respondents confirmed this; *e.g.*, R120 said, [CD lets them] “*ship as early as possible*”.

[M5] **CD lets us catch errors earlier to minimize failures.** By interviewing developers, Hilton *et al.* [27] found that the biggest perceived benefit of CI is early bug detection. So we would expect that CD also helps catch errors earlier to prevent the deployment of broken code. Our survey responses confirmed this. E.g., R51 said,

Table 1: Developers' motivation for doing CD. N=140.

Motivation	Total	Perc.*
Helps us deploy automatically instead of doing it manually	60	42.9%
Gives us smoother and easier deployments	27	19.3%
Allows us to keep our production reliable	22	15.7%
Makes releasing faster	14	10.0%
Lets us catch errors earlier to minimize failures	11	7.9%
Can enforce a deterministic workflow	11	7.9%
Lets us spend less time on maintenance and configuration	9	6.4%
Enhances the testing and validity checking	8	5.7%
Allows us to share our work and get continuous feedback	4	2.9%

*One answer may contain multiple codes; percentages need not add up to 100.

[CD helps them] "*find the problem as soon as possible. Avoid building break that other developer won't be affected*".

[M6] **CD can enforce a deterministic workflow.** Schermann *et al.* [57] found that deployment workflows, *i.e.*, structuring the release process into multiple and consecutive phases, is widespread. Their study reported that 68% of the survey respondents use the same CD workflow for dealing with issues as for every other change. And 74% of respondents agreed that they would release more frequently than they actually do by following a specific workflow. E.g., R115 said, [CD gave them] "*deterministic workflow*" and R18 said, [they want to use CD] "*because it can highly import the release phase. Change in one place will take an effort on other systems*".

[M7] **CD lets us spend less time on maintenance and configuration.** Developers perceive that not treating configuration like code leads to a significant number of production issues [47]. Developers used to spend 20% of their time setting up and fixing their test environments. CD can automatically set up the environments [8], which allows developers to spend their effort and time on more valuable activities. As R64 said, [they use CD because they want] "*to optimize time, focusing on developing the actual application*".

[M8] **CD enhances the testing and validity checking.** Mantyla *et al.* [38] analyzed the effects of moving from traditional to rapid releases on Firefox's system testing. Their study revealed that CD allows less time for testing activities but enables fast and thorough investigation of software features. Not surprisingly, this was a common theme among our survey respondents. E.g., R78 noted that [CD] "*helps with testing, less hassle*".

[M9] **CD allows us to share our work and get continuous feedback.** Krusche *et al.* [35] reported that with CD, customers can evaluate the enhancements and provide feedback immediately and in a continuous way, which improves communication between the company and its customers. Continuous feedback lets developers spend time developing the right things rather than correcting mistakes in functionality [46]. In our survey, R167 answered that, "*It helps me share my work with other contributors easily*", and R156 pointed out that [under CD, they can get] "*fast, reliable feedback*".

Developers report doing CD to reduce work, cost, and time spent on maintenance and configuration. They also report that CD helps them guarantee quality, consistency, reliability, and enhances their development process.

3.3 Tools and workflows

We asked developers which Docker workflows they subscribe to in their current CD workflow. Most report either the Docker Hub auto-builds (DH; 44.1%) or the CI-based (CI; 34.5%) workflows (Figure 1):

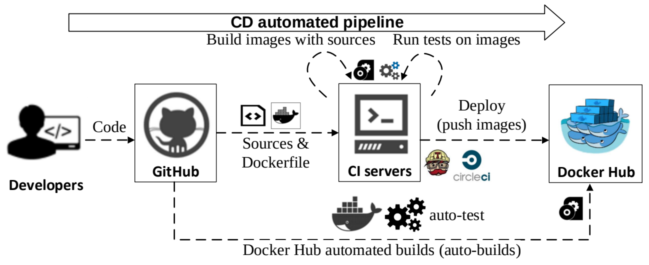
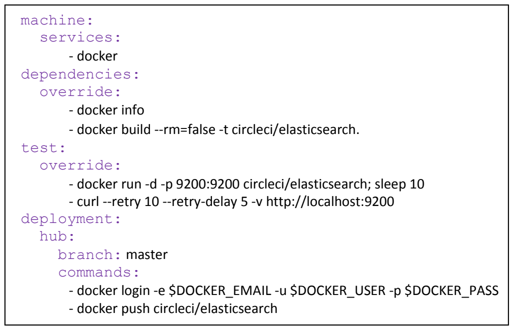


Figure 1: Overview of Docker-enabled CD workflows.



```
machine:
  services:
      - docker
dependencies:
  override:
      - docker info
      - docker build --rm=false -t circleci/elasticsearch.
test:
  override:
      - docker run -d -p 9200:9200 circleci/elasticsearch; sleep 10
      - curl --retry 10 --retry-delay 5 -v http://localhost:9200
deployment:
  hub:
    branch: master
    commands:
      - docker login -e $DOCKER_EMAIL -u $DOCKER_USER -p $DOCKER_PASS
      - docker push circleci/elasticsearch
```

Figure 2: An example of Docker settings in CircleCI.

(1) ***DH Workflow (DHW)***: Using automated builds [29] (auto-builds), Docker Hub can automatically build images from GitHub source files and push them to the corresponding Docker Hub repository. When setting up auto-builds, developers create a list of branches and tags they want to include in the Docker images. When they push code to a source code branch for one of those listed image tags, the push uses a webhook to trigger a new build, which produces a Docker image. The built image is then pushed to Docker Hub.

(2) ***CI Workflow (CIW)***: Developers automatically build images from source code using docker commands inside their CI builds; the built image is then pushed to Docker Hub. The CI tools themselves mostly integrate Docker services which, in turn, allow developers to use docker commands to build and deploy images. Figure 2 shows an example of a `circle.yml` file that specifies the standard ElasticSearch Docker image and deploys it to Docker Hub in CircleCI. CircleCI pre-installs the Docker Engine in the Linux build images, as specified in `circle.yml`. Then developers can use the "`docker build`" and "`docker push`" commands to build and deploy images.

We also asked respondents who use CIW which specific tools they use. The top-3 most frequently used CI tools are Travis CI (65.5% of respondents), Jenkins (24.1%), and CircleCI (17.2%).[7] We also found that, unexpectedly, 29.3% of respondents used two or more CI tools *simultaneously*. The reasons mentioned were generic, with the most common being that each CI tool has its respective jobs or target projects that it is good for. E.g., R39 said, "*You have to use the part of CI tools that works best for you. ... So it is really about which combination allows the best management*". Future work should examine the interplay between these seemingly equivalent and competing tools using qualitative methods.

In addition, 21.4% of respondents use other workflows. We asked them what their CD workflow consists of, and coded their answers

[7] In GitHub report [53]: the Top-3 CI tools are Travis CI, CircleCI, and Jenkins.

Table 2: Other workflows. N=36.

Other workflow	Total	Perc.
Deploy by using other services or software tools	12	33.3%
Use custom scripts	9	25.0%
Use both the DH and CI workflows	8	22.2%
Automatically test with CI but manually build and deploy	7	19.4%

into 4 groups, summarized in Table 2:
[O1] **Deploy by using other services or software tools**. For example, R17 told us their CD workflow is "*a redhat open shift service which is running based on kubernetes*";
[O2] **Use custom scripts**. For example, R48 said their CD workflow comprised "*shell scripts written by me*";
[O3] **Use both the DH and CI workflows**. E.g., R74 answered, "*I have a base image that is auto-built from a different repo by docker-cloud, and the main image which is built by CircleCI after a green build, and pushed to docker hub*";
[O4] **Automatically test with CI but manually build and deploy**. E.g., R53 told us their CD workflow is "*Push changes→Pull request→Travis confirm config it's ok→Manual builds in Docker Hub after QA in my current company*".

There is large uniformity in Docker-enabled CD workflows, with two prevalent workflows: Docker Hub auto-builds (pull-based) and continuous integration (push-based).

3.4 Unmet needs

We asked developers about unmet needs and pain points with their current CD workflows (open ended, 83 valid responses), and found that 89.9% of respondents are satisfied with their current workflow. We coded the remaining answers as listed in Table 3:
[N1] **Quicker build speed and higher throughput.** Like R4 told us, in their CD workflow "*one dockerfile takes more than 2 hours to build and timeouts*". The CD processing speed will affect the software release and developers' work efficiency. As more tests are written and more artifacts are added, the image build latency is likely to increase. 21.3% of respondents experienced increasing processing latency in their CD workflows over time, and 17.7% would change their workflow because of the increasing latency.
[N2] **Easier to learn and config.** While Docker Hub and CI tools offer a great deal of flexibility in how they can be used, this flexibility still requires a large amount of configuration even for a simple workflow. Like R109 told us, "*sometimes, circleCI config and setup is pain. Docs sometimes doesn't help*". Also, complex configuration would affect the developers collaboration, like R130 said, "*It may spent some time to teach your partner use the CD pipeline*".
[N3] **Better build testing support.** R27 told us, in their CD workflow, "*Build testing is quite a pain. I had playing docker to build OpenCV + NodeJS + Cairo which break while building the image. The build process can take up to 20mins. If it's break, I need to try other configuration and rebuild again*". As we known, CI tools provide good test integration, thus the build testing needs to be enhanced for projects that use DH workflow. Like R62 said, "*Probably I need a Jenkins or Travis container in the chain to produce more code control using some unit testing*".
[N4] **Better multi-platform build support.** Like R19 suggested, "*For applications, it's more important to provide multi-platform build*".

Table 3: Developers' unmet needs. N=83.

Needs	Total	Perc.
Quicker build speed and higher throughput	18	21.7%
Easier to learn and config	14	16.9%
Better build testing support	12	14.5%
Better multi-platform build support	11	13.3%
More features and tools integrated	11	13.3%
More flexibility and control	7	8.4%
Better support for getting info about failures and logs	6	7.2%
Better security and access controls	6	7.2%

Multi-platform build support is to meet the specific needs of different software development. Like R79 said, "*Docker hub support only x86_64 platform only. I hope that ARM support, like raspberry pi, will be added in the future. ...*".
[N5] **More features and tools integrated.** Some respondents would like their CI/CD system to integrate with more features and tools. Like R81 said, "*Docker cache still not supported without big hacks on most CI suites (e.g. Travis)*".
[N6] **More flexibility and control.** Like R147 told us, "*at some point (we) will want to use a pipeline with more control and flexibility*". In some CD workflows, there is still lack of flexibility of builds, *i.e.*, the Dockerfile optimization. Like R94 said, "*Its mainly a Docker-related drawback: I would love to be able to build Dockerfiles that have multiple images as base blocks (e.g. FROM Java8, Redis). In our workflow we always end-up copying dockerfiles from other sources and merging them in one.*".
[N7] **Better support for getting info about failures and logs.** When CD fails, developers need to identify why their CD failed. Better logging and storing test artifacts would make it easier to examine failures. But the current CD platforms are still lack of better support. Like R143 said, "*DockerHub doesn't give a whole lot of detail vs. some other solutions (CodeShip, etc)*". R60 told us their need is "*getting info about failures and debugging of broken build*".
[N8] **Better security and access controls.** Since CD workflows have access to the entire source code of a project, security and access controls are vitally important. Shu *et al.* [59] reported that more than 80% of Docker Hub images have at least one high severity level vulnerability. Like R111 told us, their pain point about CD is the "*lack of automatic security upgrades*".

Developers would like their CD workflows to be both speedy and simple to setup and maintain. This can cause some tension, since adding configurability tends to increase complexity and simplification may reduce flexibility.

Based on the previous discussion, we expect that unavoidable complexity increases over time in CD workflows would slow down the developers' workflows. We hypothesize:

H$_1$. *Image release frequency tends to decrease over time.*

In addition, a lack of better testing and debugging support would cause developers to write more test scripts or add more complexity to their build configurations, making their image build processes burdensome. So we hypothesize:

H$_2$. *Image build latency tends to increase over time.*

With more experience doing CD should also come more stability of the Docker image configurations (*i.e.*, the Dockerfile). After the initial configuration, developers may need fewer additional changes or improvements to their Dockerfiles over time. We hypothesize:

H$_3$. *Dockerfile stability tends to increase over time.*

3.5 Workflow evolution

We also asked participants about changes to their CD workflows over time (open ended, 71 answers). Among the respondents, 45.8% report having changed their CD workflows at least once before, with the common reasons (barriers) given being listed in Table 4.

[B1] **Difficult to setup and maintain.** Configuration and maintenance costs cause many developers to change their workflows. For example, R119 switched their workflow because "*the old CD pipeline is a little harder to setup. It was necessary to write several scripts to get everything working properly. The new CD pipeline is easier to setup and maintain*".

[B2] **Missing features I need.** Like R37 described, their old workflow was "*too slow and missing features*"; the poor feature support made some developers switch to a different workflow.

[B3] **Weak support for automation.** Some developers changed their workflow because their old workflow had weak support for automation. Like R128 told us, their old workflow contained "*many manual steps prone to errors*", while with the new workflow "*everything goes smoothly*".

[B4] **Overly long build times.** As we found earlier when asking about CD needs, build speed affects the developers' work efficiency. Some developers changed their workflow due to slow image build speed. For example, R159 told us, in their old CD workflow, "*Cache was dropped when the build executed, and for no good reason. So it took too much time to build the image*".

[B5] **More friction and failures.** Brittle builds make the workflow unreliable, which cause some developers to change their workflow. For example, R66 said, "*we noticed that the builds are not really reliable since there wasn't any testing. So we refactored the workflow to TravisCI which is way better for testing, but has disadvantages in speed of pushing and handling images on Docker Hub. But with a little bit of scripting the problems went away*".

[B6] **Experimenting with new tools.** Some developers changed their workflow because they wanted to use some new tools. Like R43 answered, "*just tried new stuff*".

[B7] **Steep learning curve.** Complex processes and unfamiliar configurations make some developers abandon their old workflow. Like R76 told, their old workflow was "*hard to learn, configure, or plain inefficient*".

Developers encountered increased complexity, increased latency, and decreased reliability in previous CD workflows. These barriers caused them to switch to new CD workflows.

In a CIW, CI tools may provide more effective testing support than in a DHW, which helps developers find more defects before deploying. But a CIW may require more complex configuration and maintenance than a DHW, which increases the likelihood of build failures. We hypothesize:

H_4. *CIW tends to have more failed builds than DHW.*

Moreover, because CIWs require more configuration, it may take more time to run these operations. We hypothesize:

H_5. *CIW tends to have longer build latency than DHW.*

3.6 Specifics of DH and CI workflows

We asked participants for more details on the perceived advantages of DHW or CIW, giving them a choice of 7 predefined answers (Table 5) plus the option to provide others. We collected 132 answers, 74 for DHW and 58 for CIW. For the DHW, the most important reason given was to **reduce the time spent on setting up** (85.2%), followed by to **deploy more frequently** (45.9%). Respondents also gave other reasons, *e.g.*, "*it's free and ready to work with GitHub projects*", "*Easy to share with other Docker users*". As for CIW, the most important reason was to **increase confidence in build quality and results** (79.3%), followed by to **reduce the time spent on setting up** (60.3%). Other reasons given were similar to the predefined answer we provided, *e.g.*, "*automated test and quality control*".

Table 4: Barriers with previous CD workflows. N=71.

Barriers	Total	Perc.
Difficult to setup and maintain	25	35.2%
Missing features I need	15	21.1%
Weak support for automation	13	18.3%
Overly long build times	10	14.1%
More friction and failures	10	14.1%
Experimenting with new tools	6	8.5%
Steep learning curve	4	5.6%

Table 5: Specific reasons for using a DH or CI workflow. N_{DH}=74, N_{CI}=58.

Reasons	DH workflow	CI workflow
Reduce the time spent on setting up	63 (85.2%)	35 (60.3%)
Deploy more frequently	34 (45.9%)	34 (58.6%)
Increase confidence in build quality and results	32 (43.2%)	46 (79.3%)
Less CD processing latency	23 (31.1%)	22 (37.9%)
Allow higher flexibility of builds	17 (23.0%)	13 (22.4%)
Create more visibility into team's workflow	16 (21.6%)	21 (36.2%)
Convenient custom settings and modifications	14 (18.9%)	17 (29.3%)

Overall, we found that CIWs may provide developers more integration tests to help them find errors easier and faster before publishing images to Docker Hub. This could make CIWs more reliable, increasing developer confidence in the build quality. But the DHW may provide developers with more automation and simpler configuration, which allows them to shift the time spent on setting up to other development activities. Also, we found some other differences in developers' goals when using the two workflows. More respondents (29.3%) thought CIWs have **convenient custom settings and modifications** than respondents using the DHW (18.9%). And more respondents (36.2%) thought CIWs **create more visibility into the team's workflow** than respondents using the DHW (21.6%).

The DH and CI workflows may differ in release frequency, build outcomes, build configuration stability, and build latency.

From the responses, CIWs may provide developers higher configurability than the DHW, which should result in more efficient CD workflows. So, we hypothesize:

H_6. *CIW tends to have higher release frequency than DHW.*

Similarly, the high configurability of a CIW may provide developers more possibilities to revise their Dockerfile configurations. We hypothesize:

H_7. *CIW tends to have lower Dockerfile stability than DHW.*

It is very intuitive for us to expect a significant difference between CIW and DHW. But within CIWs, different CI tools should have similar functions and roles in the CD workflow. Thus, we hypothesize:

H_8. *Within CIWs, there should not be significant differences between different CI tools.*

4 LARGE-SCALE QUANTITATIVE STUDY

Based on the findings and hypotheses from our qualitative study (detailed above, in §3), we conducted a quantitative study to explore differences between the two CD workflows (**RQ2**).

4.1 Methods

Projects selection. From the container list in Docker Store, we collected basic information for all containers listed on or before July 2017. Our survey responses show that the two widely used CD workflows are DHW and CIW (§3.3); we selected projects that used only those two workflows. For projects using CIW, we limited our study to two cloud-based CI platforms, Travis CI and CircleCI, since in the Docker Hub documentation and in our survey, we found these to be among the most popular three; the third, Jenkins, runs locally and thus has no publicly-available data or API. We call projects that use the DH workflow ***DH projects***, those that use the Travis CI workflow ***Travis projects***, and those that use the CircleCI workflow ***Circle projects***. We identified DH projects by checking for the presence of the string "`is_automated`" through the Docker Hub API (True means the project has auto-builds); this yielded 500 DH projects. For Travis and Circle projects, we identified them by checking the Docker-enabled deployment settings [12, 13], *e.g.*, "`docker push`" and "`docker build`", in their "`.travis.yml`" or "`circle.yml`" configuration files; this yielded 282 Travis projects and 343 Circle projects; each of them only used one type of CD workflow in their history.

Data collection and filtering. Out data collection involved mining three types of sources: (1) Docker Hub data, *i.e.*, Docker Hub builds, using the Docker Hub API; (2) GitHub data, *i.e.*, commits and git logs of Dockerfile, using the GitHub API; and (3) CI data, *i.e.*, CI builds, using the Travis CI and CircleCI APIs. For CI builds, the main work done by the CI tools is integration testing, so we parsed the CI build scripts[8] to distinguish between ***deployment builds*** (the aim of this CI build is to deploy images) and general test builds. We only consider deployment builds in our study. Then, we filtered out projects with less than 10 successful builds, as these might indicate experiments with the infrastructure rather than more serious CD practice. After this filtering, we obtained our final set of 855 projects for the quantitative study, 428 of them DH projects, 236 Circle projects, and 191 Travis projects.

In total, our dataset contains 133,593 image builds. Among them, 39,094 (29.3%) are Docker Hub builds, 30,990 (23.2%) are Travis CI builds and 63,509 (47.5%) are CircleCI builds. Table 6 presents aggregate descriptive statistics over the 855 projects.

Regression analysis. To test our hypotheses, we built four mixed-effects linear regression models (packages `lme4` and `lmerTest` in R) with the same random-effect term for the *base image*. The base images specified in the Dockerfile (defined in the `FROM` instruction) can give a first indication of what it is that the projects use Docker for [17]. Every Docker image starts from a base image, *e.g.*, Ubuntu base image. So we expected that the base image has an important effect on the image build process, especially the build latency. We captured the base image information by extracting its name from the specification, *i.e.*, a tuple of the form `namespace/name(:version)`.

[8]Check if the script has "`docker build`" and "`docker push`" commands.

Table 6: Aggregate statistics of the 855 projects.

Group	Statistic	Mean	St. Dev.	Min	Median	Max
DH projects	#Total builds	91.3	155.7	11	30	1,000
	#Successful builds	80.5	144.4	10	25	942
	#Errored builds	10.9	43.1	0	3	783
Travis projects	#Total builds	162.3	341.5	12	51	3,366
	#Successful builds	121.8	270.1	10	42	2,799
	#Errored builds	40.4	82.8	0	12	567
Circle projects	#Total builds	269.1	790.8	14	63	7,506
	#Successful builds	172.3	506.3	10	34	5,494
	#Errored builds	96.9	363.8	1	31	4,133

The random effect allows us to avoid modeling each base image separately, which would use up degrees of freedom unnecessarily, but still capture base image variability in the response. All other variables were modeled as fixed effects. We divide the build data of each project into different stages, in 30-day windows.

The following outcomes, or dependent variables, were observed during those time-windows:

- ***nSuccessBuilds***: number of successful builds per time window, as a proxy for release frequency.
- ***nErrorBuilds***: number of errored builds per time window, as a proxy for build results.
- ***nDockerfileChanges***: number of Dockerfile changes per time window, as a proxy for configuration stability of builds.
- ***avgBuildLatency***: mean latency of successful builds per time window, as a proxy for build speed. Build latency is the time duration from build start to end, in minutes.

Our independent variables come from two covariate areas: global (or aggregate level) and local (or time-window level):

- ***totalCommits*** and ***totalBuilds***: total number of commits and total number of image builds in the project's history, as a proxy for project size/activity.
- ***ageAtCD***: project age at the time of adopting CD, in days, computed since the earliest recorded image build.
- ***workflow***: different types of CD workflows, we distinguished DH workflow, Travis CI workflow, and CircleCI workflow. We used effect coding [26] to set the contrasts of this three-way factor, *i.e.*, comparing each level to the grand mean of all three levels.
- ***timeFlag***: label of the time window, in months, computed since the earliest image build.
- ***nLinesOfDockerfile***: number of lines of Dockerfile per time window. We removed the blank lines and comments.
- ***nIssuesOfDockerfile***: number of quality issues of the Dockerfile per time window, computed by Dockerfile Linter [17, 37].

In our models, where necessary we log-transformed dependent variables to stabilize their variance and reduce heteroscedasticity [19]. We also removed the top 1% of the data for highly-skewed variables to control outliers and improve model robustness, in line with best practices [48]. The variance inflation factors, which measure multicollinearity of the set of predictors in our models, were safe, below 3. For each model variable, we report its coefficients, standard error, significance level, and sum of squares (via ANOVA). Because each coefficient in the regression amounts to a hypothesis test, we employ multiple hypothesis correction over all coefficient results, to correct for false positives, using the Benjamini-Hochberg step-down procedure [5]. We consider the such corrected coefficients noteworthy if they were statistically significant at p<0.05. Model fit was evaluated using a marginal (R^2_m) and a conditional (R^2_c) coefficient of determination for generalized mixed-effects models [33, 44]. R^2_m describes the proportion of variance explained by the fixed effects alone, and R_c^2 describes the proportion of variance explained by the fixed and random effects together.

4.2 Results

Difference in release frequency. First, we examined the release frequency in terms of the number of successful builds per 30-day time-window. Table 7 shows the results of the release frequency model (**Model-1**). The fixed-effects part of the model explained R_m^2=0.28 of the deviance. A considerable amount of variability is explained by the random effect (R_c^2=0.43), *i.e.*, base image, not explicitly modeled by our fixed effects.

Among the fixed effects, as expected, ***totalCommits*** and ***totalBuilds*** have significant, positive effects, together explaining 64% of the variance. Thus, *big or active projects may associate with higher release frequency.* We also note that ***timeFlag*** has a significant, negative effect on release frequency (24% of the variance explained). This indicates that release frequency tends to decrease over time, holding all other variables constant, which offers support for $\mathbf{H}_1$. Compared to the overall mean across all workflows, the DH workflow has a significant negative effect on release frequency (11% of the variance explained), while the Travis CI and CircleCI workflows have significant positive effects. Holding all other variables constant, DHW tends to have lower release frequency than CIW. This is consistent with $\mathbf{H}_6$. Thus,

Release frequency tends to decrease over time. But DHW tends to have lower release frequency than CIW.

Difference in build results. Next, we used the number of errored builds to compare the workflows. Table 8 shows the summary of the build results regression model (**Model-2**). The fixed-effects part of the model explained R_m^2=0.21 of the deviance, for a total R_c^2=0.27 with the random effect.

Among the model results, ***totalCommits*** and ***totalBuilds*** have a significant positive effect,[9] but they explain different proportions of variance (40% vs 4%), consistent with *more code may bring more errors to the build.* ***timeFlag*** has a significant negative effect (12% of the variance explained). Thus, *the number of errored builds tends to become smaller over time.* Compared to the overall mean of three workflows, the DH workflow has a significant negative effect (37% of the variance explained). While the Travis CI and CircleCI workflows have significant positive effects on the outcome. This indicates that compared to DHW, CIW is associated with more errors in the builds, holding all other variables constant; this is consistent with $\mathbf{H}_4$. Thus,

CIW tends to have more errored image builds than DHW.

Difference in build configuration stability. We next used the number of Dockerfile changes per 30-day time-windows to compare the build configuration stability between the workflows; thus, higher # changes indicates lower stability. Table 9 shows the configuration stability model (**Model-3**) summary. The fixed-effects explained R_m^2=0.07 of the deviance; with the random effect the model explained R_c^2=0.13 of the deviance, and this was our poorest fitting model.

[9]Note that in this model, "positive" effect means more errored builds and "negative" effect means fewer errored builds.

Table 7: Release frequency model. The response is *log(nSuccessBuild)*. R_m^2=0.28, R_c^2=0.43.

	Coeffs (Error)	Sum Sq.
(Intercept)	0.3631 (0.0398)***	
totalCommits	0.4041 (0.0202)***	265.84***
ageAtCD	-0.0602 (0.0156)***	9.89***
totalBuilds	0.4625 (0.0160)***	551.21***
timeFlag	-0.0385 (0.0018)***	306.15***
nIssuesOfDockerfile	-0.0229 (0.0142)	1.72
nLinesOfDockerfile	-0.0359 (0.0138)*	4.44**
workflow=DH	-0.3244 (0.0219)***	145.76***
workflow=Travis CI	0.1922 (0.0228)***	
workflow=CircleCI	0.1322 (0.0213)***	

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table 8: Build results model. The response is *log(nErrorBuild)*. R_m^2=0.21, R_c^2=0.27.

	Coeffs (Error)	Sum Sq.
(Intercept)	0.0806 (0.0382)*	
totalCommits	0.3754 (0.0241)***	184.56***
ageAtCD	-0.1426 (0.0227)***	30.21***
totalBuilds	0.1135 (0.0238)***	17.32***
timeFlag	-0.0243 (0.0028)***	56.03***
nIssuesOfDockerfile	-0.0315 (0.0209)	1.74
nLinesOfDockerfile	-0.0058 (0.0210)	0.06
workflow=DH	-0.4566 (0.0332)***	169.40***
workflow=Travis CI	0.1293 (0.0298)***	
workflow=CircleCI	0.3273 (0.0271)***	

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table 9: Configuration stability model. The response is *log(nDockerfileChanges)*. R_m^2=0.07, R_c^2=0.13.

	Coeffs (Error)	Sum Sq.
(Intercept)	0.2257 (0.0401)***	
totalCommits	0.1193 (0.0235)***	23.11***
ageAtCD	-0.0845 (0.0253)**	9.97***
totalBuilds	-0.0027 (0.0222)	0.01
timeFlag	-0.0346 (0.0035)***	87.40***
nIssuesOfDockerfile	0.0479 (0.0235)	3.72*
nLinesOfDockerfile	0.0775 (0.0240)**	9.35**
workflow=DH	-0.1668 (0.0326)***	27.93***
workflow=Travis CI	0.0350 (0.0368)	
workflow=CircleCI	0.1318 (0.0331)***	

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

timeFlag has a significant negative effect, accounting for 54% of the variance explained. Holding all other variables constant, it indicates that the build configuration tends to become more stable (*i.e.*, have fewer changes) over time; this is consistent with $\mathbf{H}_3$. Compared to the overall mean of three workflows, the DH workflow has a significant negative effect (17% of the variance explained), but the Travis CI workflow has no significant difference; the CircleCI workflow has a significant positive effect. This is evidence that CIW may have lower Dockerfile stability than DHW, holding all other variables constant. Hence, this is consistent with $\mathbf{H}_7$.

Container configuration stability tends to increase over time. But CIW tends to have lower Dockerfile stability than DHW.

Difference in build latency. Finally, we examined build latency. Table 10 shows the build latency model (**Model-4**) result. The fraction of total deviance explained by the fixed-effects part of the model is R_m^2=0.20. A considerable amount of variability is explained by the random effect (R_c^2=0.57). This is consistent with our description in §4.1, of the strong effect the base image latency may have on the total build latency.

Table 10: Build latency model, The response is *log(avgBuildLatency)*. R^2_m=0.20, R^2_c=0.57.

	Coeffs (Error)	Sum Sq.
(Intercept)	-0.1974 (0.0544)***	
totalCommits	0.1551 (0.0170)***	44.56***
ageAtCD	0.0139 (0.0142)	0.52
totalBuilds	0.2023 (0.0148)***	99.75***
timeFlag	0.0110 (0.0016)***	24.39***
nIssuesOfDockerfile	0.0387 (0.0131)**	4.70**
nLinesOfDockerfile	0.1381 (0.0127)***	63.07***
workflow=DH	0.4336 (0.0204)***	245.90***
workflow=Travis CI	-0.2891 (0.0209)***	
workflow=CircleCI	-0.1445 (0.0196)***	

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

As expected, more lines in a Dockerfile (***nLinesOfDockerfile***) are associated with longer build latency (13% of the variance explained). ***timeFlag*** has a small, significant positive effect (5% of the variance explained), meaning build latency tends to increase over time, holding other variables constant; this is consistent with $\mathbf{H}_2$. Compared to the overall mean across all workflows, the DH workflow has a strong, positive effect (51% of the variance explained), and the Travis CI and CircleCI workflows have significant negative effects. This means that DHW tends to have longer build latency than CIW, holding all other variables constant, which is contrary to our expectation. So, $\mathbf{H}_5$ is rejected.[10]

Build latency tends to increase slightly over time. Interestingly, DHW tends to have longer build latency than CIW.

Differences among CI workflows. From our models, we find that usage of CIW or DHW associate with significant differences in outcomes, consistent with our hypotheses. But with respect to errored builds and build latency, we also find differences between the Travis CI and CircleCI workflows. This indicates that different CI tools may perform the same or similar role, but be associated with different effects. So $\mathbf{H}_8$ is rejected. Thus,

DHW and CIW are significantly different. Using different CI tools can also associate with different outcomes.

5 DISCUSSION

Here we discuss the practical differences and the trade-offs between the DH workflow (DHW) and CI workflows (CIWs), followed by the practice implications.

5.1 Practical differences

We recapitulate the practical differences between DHW and CIW based on their support for CD automation and Docker, build environment, as well as developer experience. These recapitulations allowed us to develop a deeper understanding of our survey and quantitative study results.

Support for automated testing. In CIW, CI tools set up "hooks" with GitHub to automatically run tests (typically unit and integration) at specified times. By default, these are set up to run after a pull request is created or when code is pushed to GitHub. DHW has a complementary automated testing tool for deployment images (auto-test) [28], provided by Docker Hub. Before using auto-test, a `docker-compose.test.yml` automated test file must be set up. This file defines a `sut` service that lists the tests to be run and it should be located in the same directory that contains the Dockerfile. Since the `docker-compose.test.yml` is a standard Compose file, developers could also just invoke Compose in the CI configuration file to run those tests, which implies CIW may be more powerful.

Support for Docker and Docker tools. DHW, naturally, has better support for Docker and Docker tools than CIW, since Docker Hub itself is a cloud-based service provided by Docker. In addition, CI tools differ in the amount of Docker support they provide. Docker version support provided in Travis CI is more recent and more diverse than that in CircleCI. Travis CI developers can manually upgrade Docker to the latest version by updating `.travis.yml` [10], whereas CircleCI currently supports only 3 fixed Docker versions [14]. Also, we found that Travis CI has some Docker tools pre-installed, *e.g.*, the Docker Compose tool [11]. In CircleCI, developers need to install and configure this tool in their container in order to use it [15].

Build environment. As reported on the Docker forum [24], the current limits on Docker Hub auto-builds are 1 CPU and 2 GB RAM, which in practice means potential latency problems for large builds. On the other hand, Travis CI and CircleCI both provide 2 CPUs and larger RAM limits (4 GB and 8 GB) for the build environment [9, 16]. We found latency to be an issue in Docker Hub in our survey and quantitative study.

Developer work experience. In our survey, we found that the average OSS work experience of respondents who use DHW is 7.8 years (median 6), while for respondents who use CIW it is 8.8 years (median 7). While CIW users seem to have one additional year of OSS experience, the difference is not statistically significant (Wilcoxon test; p=0.42). On the other hand, the average CI/CD work experience of DHW respondents is 3.6 years (median 3), and of CIW respondents it is 4.5 years (median 4). The statistical test shows that this difference is significant (Wilcoxon test; p=0.04). So, in practice this may mean that the use/implementation of CIW associates with more developer CI/CD experience than DHW.

5.2 Trade-Offs between CD workflows

Our survey and data analysis revealed that when choosing between CIW and DHW one may have to trade some features for others and that it is unlikely that one workflow will fit all:

Higher configurability (CIW) vs. Higher simplicity (DHW) (see M7, B1, N2, N6, Model-3 and §5.1). Highly configurable workflow means also one that is harder to use due to its complexity. On the other hand, simplicity means higher build configuration stability, but may also mean less control and lower flexibility.

Higher performance (CIW) vs. Diverse needs (DHW) (see N1, B2, B3, B4, Model-1, Model-4 and §5.1). Specific requirements (*e.g.*, different Docker versions) may bring about lower performance. But higher performance may not meet more diverse needs.

Higher reliability (CIW) vs. Lower maintenance (DHW) (see M3, M5, M8, N3, B5, Model-2 and §5.1). More testing means higher reliability, but also more errored builds, *i.e.*, more maintenance.

[10]Some developers have posted about the build latency problem of Docker Hub auto-builds on the Docker forum (https://forums.docker.com/t/why-does-it-take-so-long-for-the-docker-hub-automated-builds-to-upload-the-built-image). Some in our survey are ok with it. R23 said, "*The latency and build times are totally okay due to the fact that it's free. I would have a different opinion if I paid for the same service though*".

Higher scalability (CIW) vs. Lower experience (DHW) (see N2, N5, B7 and §5.1). Higher scalability means that more tools or services can be integrated and do not strongly limit the CD process (*e.g.*, build speed) in a project, even if the image grows in size and complexity. But CIW may require more experienced developers.

5.3 Implications

For researchers. Our studies provide a rich resource of initial ideas for further study. Our survey showed that 45.8% of the respondents have changed from one to another CD workflows (§3.5). Examining the costs and benefits that arise from switching CD workflows may point to best practices for developers needing to change their solutions. Hence, our study motivates future work to explore the CD workflow evolution.

We also found that developers have a choice of different CI tools (§3.3), and that using different CI tools associates with different outcomes (§4.2). Therefore, researchers should investigate the barriers and benefits developers face when using particular CI tools.

Our quantitative study mostly focused on comparing CD outcomes between DHW and CIW (§4). How the two CD workflows differ in other dimensions should be further empirically evaluated. E.g., we have found in our data, only anecdotally, that some types of projects may gravitate toward one or the other of the workflows. With much more data and careful project classification along different dimensions, some patterns may become apparent. Our findings motivate the need for collecting more empirical evidences that help developers, who wish to reduce complexity and improve performance, to choose appropriate CD tools and establish CD workflow without arbitrary decisions.

For developers. Our study shows that developers face trade-offs when choosing different CD workflows (§5.2). A direct implication is that developers should not only consider their own experiences and needs, but also consider the different CD support in the CD workflows. E.g., less experienced developers may benefit more from using DHW instead of CIW, because DHW has higher simplicity and lower maintenance cost. More experienced developers may instead benefit from CIW, which can bring higher configurability and performance. Hence, there is a need for a list of "bespoke CD best practices" for developers with different experiences and needs.

Our quantitative studies revealed that Dockerfile configuration details, *e.g.*, base image, have important effects on the CD workflow outcomes (§4.2). Developers should select appropriate base image and instructions. Also, developers should simplify their Dockerfile content and optimize the image structures, *i.e.*, image layers and instruction orders. Therefore, the issue of how to manage and help developers configure the best Dockerfile needs to be addressed.

For service providers. Based on the trade-offs developers face (§5.2), there are two suggestions for service providers, one is simplifying the configuration complexity, to lower the initiation obstacles for more inexperienced developers. The other is improving their support for CD automation, *e.g.*, integrating more powerful Docker tools and providing more virtual machine environments.

For tool builders. Our respondents expressed their needs for build testing and failure logging (N3 and N7). Hence, tool builders may look into creating modern tools that enhance build testing and integrate with different workflows. Also, developers could use more sophisticated (*e.g.*, social coding integrated/enabled) tools that manage and analyze logs of build failures.

6 THREATS TO VALIDITY

Internal validity. Surveys can be affected by bias and inaccurate responses, which may be intentional or unintentional. To ameliorate this threat, we designed and delivered our survey by following established guidelines [51, 61]. Most of our questions are open-ended so that participants can freely fill their own answers, and during our manual analysis, we carefully removed unrelated answers.

In the quantitative study, we controlled for the build complexity with the number of lines in the Dockerfile, and we set the base image as a random-effect. But the Dockerfile may have many different instructions inside, which may cause some bias, although in our manual examination we did not find evidence for it. We note that our models' fit to the data is around 25% of the deviance, and lower for the build configuration stability model (Model-3). That is not necessarily a problem for our purposes as we are only interested in the coefficients' effect and not relying on the models to explain the full phenomena, which would require many more variables, and is beyond the scope of this work.

External validity. We only considered Docker repositories that are on GitHub. Thus, our findings cannot be assured to generalize to projects hosted on other services, *e.g.*, Bitbucket and GitLab, although there is no inherent reason why they would be biased. Moreover, we only analyzed open source software. CD workflow and its influence might be different in closed source environments. Finally, we conducted our study on the Docker-enabled CD workflow. We cannot assume that our findings generalize to other CD workflows that are not using Docker.

7 CONCLUSION

We conducted the first large-scale study of Docker-enabled CD workflows on Docker Hub/GitHub to shed light on the developers' experiences and expectations when using CD. Our mixed qualitative and quantitative approach enabled us to tease out categories of developers' opinions on CD and the two workflows, as well as test hypotheses arising from them using large data sets. Most of our survey findings were confirmed in the data, but some were not, emphasizing the power of mixed methods to produce holistic findings. Our findings indicate that developers face trade-offs when choosing between different CD workflows with respect to configurability, simplicity, requirements, performance, stability, developer experience, *etc.*, and we were able to distill some implications for different stakeholders.

ACKNOWLEDGMENTS

The bulk of this work was produced while the first author was visiting DECAL at UC Davis. We thank members of DECAL, especially Prof. Devanbu, for their comments and directions on this research. We also thank the 168 survey respondents for their valuable answers and the anonymous reviewers for their insightful comments on earlier versions of this paper. This work was supported by the NSF (Grants No. 1717370 and 1717415), the National Natural Science Foundation of China (Grant No. 61502512 and 61432020), and China Scholarship Council.

REFERENCES

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

[2] Valentina Armenise. 2015. Continuous delivery with Jenkins: Jenkins solutions to implement continuous delivery. In *International Workshop on Release Engineering (RELENG)*. IEEE, 24–27.

[3] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

[4] Robert Benefield. 2009. Agile deployment: Lean service management and deployment strategies for the SaaS enterprise. In *Hawaii International Conference on System Sciences (HICSS)*. IEEE, 1–5.

[5] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the royal statistical society. Series B (Methodological)* (1995), 289–300.

[6] David Bernstein. 2014. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.

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

[8] Lianping Chen. 2015. Continuous delivery: Huge benefits, but challenges too. *IEEE Software* 32, 2 (2015), 50–54.

[9] Travis CI. 2018. Build Environment Overview. Retrieved July 17, 2018 from https://docs.travis-ci.com/user/reference/overview/

[10] Travis CI. 2018. Installing a newer Docker version. Retrieved July 17, 2018 from https://docs.travis-ci.com/user/docker/#Installing-a-newer-Docker-version

[11] Travis CI. 2018. Using Docker Compose. Retrieved July 17, 2018 from https://docs.travis-ci.com/user/docker/#Using-Docker-Compose

[12] Travis CI. 2018. Using Docker in Builds. Retrieved July 17, 2018 from https://docs.travis-ci.com/user/docker/

[13] CircleCI. 2018. Continuous Integration and Delivery with Docker. Retrieved July 17, 2018 from https://circleci.com/docs/1.0/docker/

[14] CircleCI. 2018. Docker version. Retrieved July 17, 2018 from https://circleci.com/docs/2.0/building-docker-images/#docker-version

[15] CircleCI. 2018. Installing and Using docker-compose. Retrieved July 17, 2018 from https://circleci.com/docs/2.0/docker-compose/

[16] CircleCI. 2018. Remote Docker Environment. Retrieved July 17, 2018 from https://circleci.com/docs/2.0/building-docker-images/#specifications

[17] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. 2017. An empirical analysis of the Docker container ecosystem on GitHub. In *International Conference on Mining Software Repositories (MSR)*. IEEE Press, 323–333.

[18] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. 2015. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology* 57 (2015), 21–31.

[19] Jacob Cohen, Patricia Cohen, Stephen G West, and Leona S Aiken. 2013. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge.

[20] T. Combe, A. Martin, and R. Di Pietro. 2016. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Computing* 3, 5 (2016), 54–62.

[21] Juliet Corbin and Anselm Strauss. 1990. Grounded theory research: Procedures, canons and evaluative criteria. *Zeitschrift für Soziologie* 19, 6 (1990), 418–427.

[22] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. 2014. Virtualization vs containerization to support paas. In *International Conference on Cloud Engineering (IC2E)*. IEEE, 610–614.

[23] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and linux containers. In *International Symposium On Performance Analysis of Systems and Software (ISPASS)*. IEEE, 171–172.

[24] Docker forum. 2015. Automated Build resource restrictions. Retrieved July 17, 2018 from https://forums.docker.com/t/automated-build-resource-restrictions/1413/2

[25] Martin Fowler and Matthew Foemmel. 2006. Continuous integration. *ThoughtWorks http://www.thoughtworks.com/Continuous Integration.pdf* 122 (2006).

[26] Alkharusi H. 2012. Categorical variables in regression analysis: A comparison of dummy and effect coding. *International Journal of Education* 4, 2 (2012), 202–210.

[27] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *International Symposium on Foundations of Software Engineering (FSE)*. ACM, 197–207.

[28] Docker Hub. 2018. Automated repository tests. Retrieved July 17, 2018 from https://docs.docker.com/docker-cloud/builds/automated-testing/

[29] Docker Hub. 2018. Configure automated builds on Docker Hub. Retrieved July 17, 2018 from https://docs.docker.com/docker-hub/builds/

[30] Jez Humble. 2018. Continuous Delivery. Retrieved July 17, 2018 from https://continuousdelivery.com/

[31] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education.

[32] Jez Humble, Chris Read, and Dan North. 2006. The deployment production line. In *Agile Conference (AGILE)*. IEEE, 6–pp.

[33] Paul CD Johnson. 2014. Extension of Nakagawa & Schielzeth's R2GLMM to random slopes models. *Methods in Ecology and Evolution* 5, 9 (2014), 944–946.

[34] Sebastian Klepper, Stephan Krusche, Sebastian Peters, Bernd Bruegge, and Lukas Alperowitz. 2015. Introducing continuous delivery of mobile apps in a corporate environment: A case study. In *International Workshop on Rapid Continuous Software Engineering (RCoSE)*. IEEE, 5–11.

[35] Stephan Krusche and Lukas Alperowitz. 2014. Introduction of continuous delivery in multi-customer project courses. In *International Conference on Software Engineering (ICSE)*. ACM, 335–343.

[36] M. Leppanen, S. Makinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V. Mantyla, and T. Mannisto. 2015. The highways and country roads to continuous deployment. *IEEE Software* 32, 2 (2015), 64–72.

[37] Linter. 2018. Dockerfile Linter. Retrieved July 17, 2018 from http://hadolint.lukasmartinelli.ch/

[38] Mika V Mantyla, Foutse Khomh, Bram Adams, Emelie Engstrom, and Kai Petersen. 2013. On rapid releases and software testing. In *International Conference on Software Maintenance (ICSM)*. IEEE, 20–29.

[39] AR Manu, Jitendra Kumar Patel, Shakil Akhtar, VK Agrawal, and KN Bala Subramanya Murthy. 2016. Docker container security via heuristics-based multilateral security-conceptual and pragmatic study. In *International Conference on Circuit, Power and Computing Technologies (ICCPCT)*. IEEE, 1–14.

[40] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[41] Mathias Meyer. 2014. Continuous integration and its tools. *IEEE Software* 31, 3 (2014), 14–16.

[42] Adrian Mouat. 2015. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, Inc.

[43] Preeth E N, F. J. P. Mulerickal, B. Paul, and Y. Sastri. 2015. Evaluation of Docker containers based on hardware utilization. In *International Conference on Control Communication Computing India (ICCC)*. 697–700.

[44] Shinichi Nakagawa and Holger Schielzeth. 2013. A general and simple method for obtaining R2 from generalized linear mixed-effects models. *Methods in Ecology and Evolution* 4, 2 (2013), 133–142.

[45] Steve Neely and Steve Stolt. 2013. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *Agile Conference (AGILE)*. IEEE, 121–128.

[46] Helena Olsson Holmström, Hiva Alahyari, and Jan Bosch. 2012. Climbing the "Stairway to Heaven" A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *Euromicro Conference on Software Engineering and Advanced Applications*. Ieee Computer Soc, 392–399.

[47] Chris Parnin, Eric Helms, Chris Atlee, Harley Boughton, Mark Ghattas, Andy Glover, James Holman, John Micco, Brendan Murphy, Tony Savor, et al. 2017. The top 10 adages in continuous deployment. *IEEE Software* 34, 3 (2017), 86–95.

[48] Jagdish K Patel, CH Kapadia, and Donald Bruce Owen. 1976. *Handbook of statistical distributions*. M. Dekker.

[49] Perforce. 2017. Continuous Delivery: The New Normal for Software Development. Retrieved July 17, 2018 from https://www.perforce.com/sites/default/files/files/2017-09/continuous-delivery-report.pdf

[50] Portworx. 2017. 2017 Annual Container Adoption Survey: Huge Growth in Containers. Retrieved July 17, 2018 from https://portworx.com/2017-container-adoption-survey/

[51] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. 2003. Conducting on-line surveys in software engineering. In *International Symposium on Empirical Software Engineering (ISESE)*. IEEE, 80–88.

[52] Puppet. 2017. 2017 State of DevOps Report. Retrieved July 17, 2018 from https://puppet.com/resources/whitepaper/state-of-devops-report

[53] GitHub report. 2017. GitHub welcomes all CI tools. Retrieved July 17, 2018 from https://blog.github.com/2017-11-07-github-welcomes-all-ci-tools

[54] Pilar Rodríguez, Alireza Haghighatkhah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M Verner, and Markku Oivo. 2017. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software* 123 (2017), 263–291.

[55] Chuck Rossi, Elisa Shibley, Shi Su, Kent Beck, Tony Savor, and Michael Stumm. 2016. Continuous deployment of mobile software at facebook (showcase). In *International Symposium on Foundations of Software Engineering (FSE)*. ACM, 12–23.

[56] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *International Conference on Software Engineering (ICSE)*. ACM, 21–30.

[57] Gerald Schermann, Jürgen Cito, Philipp Leitner, Uwe Zdun, and Harald Gall. 2016. *An empirical study on principles and practices of continuous delivery and deployment*. Technical Report. PeerJ Preprints.

[58] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943.

[59] Rui Shu, Xiaohui Gu, and William Enck. 2017. A Study of Security Vulnerabilities on Docker Hub. In *International Conference on Data and Application Security and Privacy*. ACM, 269–280.
[60] John Ferguson Smart. 2011. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. O'Reilly.
[61] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. 2013. Improving developer participation rates in surveys. In *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 89–92.
[62] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. 2017. A Tale of CI Build Failures: an Open Source and a Financial Organization Perspective. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–193.
[63] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. 2016. Continuous delivery practices in a large financial organization. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 519–528.
[64] David Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2018. I'm Leaving You, Travis: A Continuous Integration Breakup Story. In *International Conference on Mining Software Repositories (MSR)*. ACM, 165–169.
[65] Tianyin Xu and Darko Marinov. 2018. Mining Container Image Repositories — MSR for Software Configuration and Beyond. In *International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. ACM, 49–52.
[66] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: A large-scale empirical study. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 60–71.