PIPELINEASCODE: A CI/CD Workflow Management System through Configuration Files at ByteDance

Xianhao Jin*, Yifei Feng*, Chen Wang†, Yang Liu‡, Yongning Hu*, Yufei Gao*, Kun Xia*, Luchuan Guo*
ByteDance Ltd.
*San Jose, CA, USA †Singapore ‡Shanghai, China
{xianhao.jin, yifei.feng, wangchen.roy, liuyang.leon, yongning.hu, yufei.gao, kun.xia, luchuanguo}@bytedance.com

***Abstract*—Continuous Integration (CI) and Continuous Deployment (CD) are widely used practices in modern software engineering. Unfortunately, it is also an expensive and complicated practice — setting up a CI/CD pipeline can be time-consuming and error-prone. CI/CD pipelines at ByteDance can be set up and modified through the user interface by dragging each atom onto the screen to maximize its customizability. However, these operations increase the difficulty of reuse and version tracking. In this work, we design PIPELINEASCODE as a CI/CD workflow management system using configuration files to provide high portability and traceability. PIPELINEASCODE is seamlessly integrated with the CI/CD system at ByteDance which is named as "ByteCycle". We also assess PIPELINEASCODE in two dimensions to determine its strengths by addressing two key questions: what types of pipelines developers prefer to manage using PIPELINEASCODE, and how pipelines evolve after the adoption of PIPELINEASCODE. The evaluation results show that pipelines managed by PIPELINEASCODE generally have a higher build frequency, fewer steps, a higher change frequency, a lower success rate and a longer build duration. After the adoption of PIPELINEASCODE, pipelines tend to have fewer steps, a higher build frequency, a longer build duration, a higher success rate and a higher change frequency. The results show that PIPELINEASCODE is capable of improving the reliability and flexibility of CI/CD pipelines and thus encourages users to build and deploy more frequently.**

***Index Terms*—continuous integration, continuous deployment, software maintenance, empirical software engineering**

I. INTRODUCTION

Continuous Integration (CI) is a software development practice by which developers integrate code into a shared repository several times a day [15] and Continuous Deployment (CD) is a software development practice that works in conjunction with CI to automate the infrastructure provisioning and application release process [20]. At ByteDance, CI/CD operations are completed on the platform of "ByteCycle" where developers can set up their customized CI/CD pipelines using different combinations of atoms (one atom is a unit service to perform a group of related tasks, allowing ByteCycle to integrate with other platforms) and trigger the build process based on their own preferences. However, the setup of the pipeline can be a very manual process, demanding numerous clicks with the mouse: developers need to find their target atoms in the atom market patiently, add the input of each atom correctly, and arrange the execution order of atoms cautiously if they want to create a pipeline from 0 to 1. This is already time-consuming, not to mention developers may also need to set the appropriate trigger schedule and make modifications to the pipeline from time to time. What is even worse, if one developer wants to create a very similar pipeline or just reuse this pipeline in some other places, the developer will have to repeat this process. In addition, once the changes on the pipeline are saved, it is very difficult to rollback it to previous versions and track the change history of this pipeline.

In this work, our goal is to automate the process of reusing the CI/CD pipelines and to improve the pipelines' version-tracking ability at ByteDance. To achieve this goal, we propose PIPELINEASCODE by taking advantage of the idea of "as code" such as infrastructure as code [47] and utilize configuration files as the definitive source for maintaining CI/CD pipelines. Once the configuration file is translated successfully, developers have the option to copy and paste it for reuse when setting up the pipeline. Then developers also gain the ability to track the modification history of the pipeline by checking the configuration file in the codebase repository. Although this concept has been successfully implemented by other existing tools [10], we still find it necessary to design our own system to integrate with our unique CI / CD platform at ByteDance and the actual performance of the idea is still unclear.

Furthermore, we provide the first empirical study to evaluate PIPELINEASCODE to understand the specific preferences of developers when it comes to managing them with PIPELINEASCODE and examine the transformations that occur in pipelines after the integration of PIPELINEASCODE. We answer the following two research questions:

RQ1: What types of pipelines do developers prefer to manage using PIPELINEASCODE?

RQ2: How do pipelines evolve after the adoption of PIPELINEASCODE?

To answer RQ1, we compare the characteristics of the pipelines that adopt PIPELINEASCODE with other pipelines that do not adopt it. To answer RQ2, we compare the characteristics of the pipelines before and after the adoption of PIPELINEASCODE. For both research questions, we used five metrics to measure the characteristics of pipelines: the duration of pipeline construction, the frequency of pipeline

to be triggered, the frequency of pipeline to be modified, the pipeline success rate, and the number of atoms included in the pipeline. These metrics can be used to reflect the complexity of the pipeline and the developer behaviors on the CI/CD pipelines.

The results of RQ1 show that pipelines managed by PIPELINEASCODE have a higher build frequency, fewer atoms, a higher change frequency, a lower success rate, and a longer build duration, in general. The findings of RQ1 imply that developers may prefer to use PIPELINEASCODE on simpler pipelines that need to be triggered and changed more often and hope that PIPELINEASCODE can make pipeline modifications easier and cause fewer failures. The results of RQ2 reflect that pipelines tend to have fewer atoms, a higher build frequency, a longer build duration, a higher success rate and a higher change frequency after the adoption of PIPELINEASCODE. The RQ2 findings confirm the benefits of our tool PIPELINEASCODE, which can allow developers to build pipelines more often, reduce the possibility of bursting the pipeline, and make pipeline modifications easier. The findings also show that the adoption of PIPELINEASCODE can improve the reliability of CI/CD pipelines by causing fewer failures and flexibility through more frequent changes in pipelines so that CI/CD users are encouraged to run pipelines more frequently to observe failures earlier and to make more frequent deployments.

II. BACKGROUND

A. Continuous Integration

Continuous integration (CI) is a DevOps practice [11] software development in which developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous integration aims to solve the problem that developers might work in isolation and check their changes only after its full completion, resulting in time-consuming merging and accumulated bugs without correction. Continuous integration benefits the software development team by improving developers' productivity, finding and addressing bugs earlier, and accelerating the delivery process. The best practices of continuous integration require developers to commit early and often. Well-known examples of CI services are Jenkins[1], Travis[2], and CircleCI[3] [22]. CI services can also be built-in in social coding platforms such as GitHub and GitLab [9]. In addition, big tech companies such as Google and Facebook have their own designed continuous integration system.

A complete CI build comprises 1) a traditional build and compile phase, 2) a phase in which automated static analysis tools (ASAT) are used, and 3) a testing phase, in which unit, integration, and system tests are run [4]. In practice, the build can include multiple jobs, and these jobs can be executed in a parallel way. Any failure occurring in any of these three phases can cause the build to fail, *i.e.,* error, or fail.

[1]https://www.jenkins.io/
[2]https://travis-ci.org/
[3]https://circleci.com/

B. Continuous Delivery and Continuous Deployment

Continuous Delivery (CDE) is a software engineering approach in which teams continue to produce valuable software in short cycles and ensure that the software can be reliably released at any time [6]. Companies that practice continuous delivery have reported great benefits, such as significant improvements in time to market, customer satisfaction, product quality, release reliability, productivity and efficiency, and the ability to build the right product through rapid experiments [7], [40]. As best practice, continuous delivery often works in conjunction with CI to automate the process of establishing the infrastructure and releasing applications. After the code has passed the phases of building and testing within the CI process, Continuous Delivery takes the reins in the final stages to package it with all the necessary components for deployment in multiple environments at any given time.

Another concept similar to Continuous Delivery is Continuous Deployment (CD), which refers to the automatic and continuously deploy of the application to target environments. What differentiates continuous deployment from continuous delivery is a production environment (*i.e.,* actual customers): the goal of continuous deployment practice is to automatically and steadily deploy every change in the production environment [60]. Continuous deployment often involves canary deployment for testing in the production and multiple control plane deployments for better performance, high availability, and high reliability.

C. ByteCycle at ByteDance

ByteCycle is an automated Research and Development (R&D) management platform including CI/CD management provided by ByteDance. It provides end-to-end R&D solutions across requirements, research and development, testing, releases, operations and maintenance, and measurements. In this way, ByteCycle empowers teams to standardize Research and Development processes and improve team collaboration efficiency for improved Research and Development efficiency. ByteCycle is a one-stop research and development management platform initiated by different teams. It connects the entire development process, from needs analysis to research, development, construction, testing, release, operation, and measurement, to address the issue of fragmented development activities caused by the dispersion of supporting platforms. Its purpose is to enhance the development experience.

The core concept of ByteCycle is pipeline, which is used to describe the workflow of the CI/CD process, and there are primarily two kinds of pipeline: free-style pipeline and project pipeline. The former is managed in the different workspaces for authorization. Freestyle pipelines are free to modify including their atoms and trigger schedules. Project pipelines are bound to the corresponding projects and have more specific purposes. For example, project pipelines can be created to deploy the application to the test environment or run in the integration test. Project pipelines can also be used to rollback the previous deployments that are also applied through pipelines. Both freestyle and project pipelines have

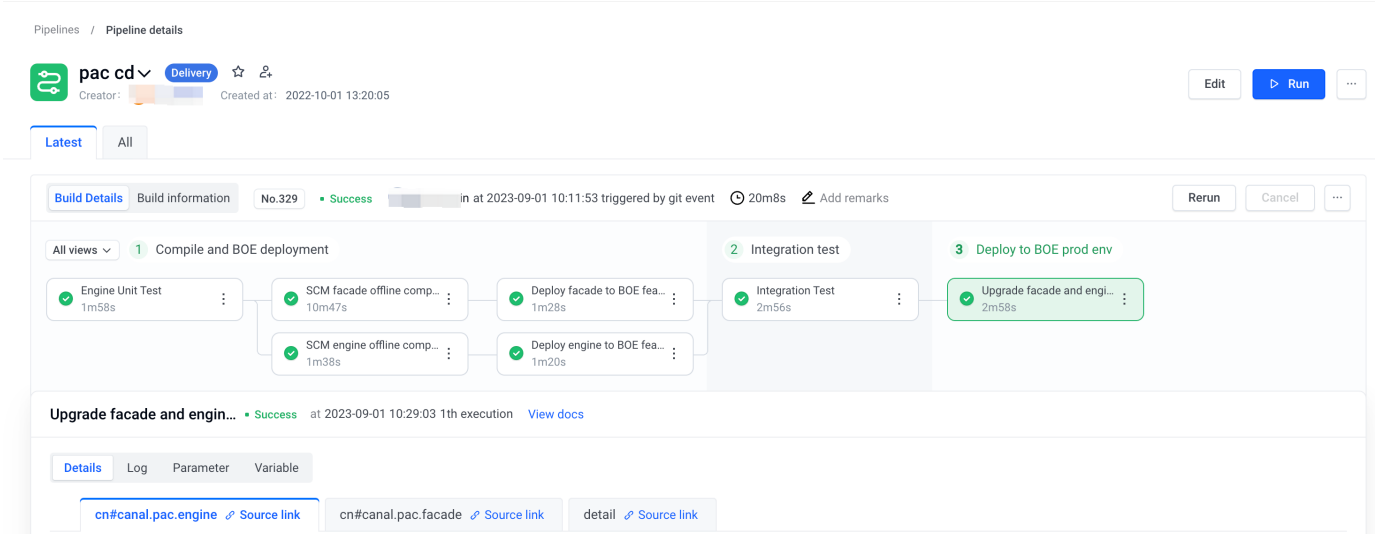

Fig. 1: Example pipeline in ByteCycle.

the same components: *e.g.,* a list of atoms to be executed in specific orders, a list of trigger conditions to describe the trigger schedule and a list of variables to set the inputs of atoms or convey values across atoms. The atom service of CI/CD pipelines is introduced as a new concept in ByteCycle. An atom is designed to complete some specific related tasks to let ByteCycle integrate with other platforms, *e.g.,* build the application, deploy it to the test environment, and perform manual confirmation. Developers can customize their own atoms and publish the atoms to the open-atom market to be used across the ByteCycle platform. ByteCycle pipelines also support sub-pipelines, which means the main pipeline depends on the sub-pipelines and will wait until the sub-pipelines are completed and can gather some information from sub-pipelines. This feature is also implemented by a specific atom called the "subpipeline driver".

Figure 1 depicts an example pipeline for deploying two related applications in the deployment environment in ByteCycle. From the screenshot, we can observe that the latest build in the pipeline provides essential information, such as the build number, trigger type, trigger time, and build duration. The pipeline has three phases, and each phase includes multiple atoms. In the first phase of "Compile and BOE deployment", the atom is added to run some unit tests. Following completion of the task, two subroutines operate simultaneously to compile the two applications and deploy them into the test environment. Then in the second phase, the integration test is conducted based on the deployed applications in the former phase. Finally, if the integration tests are passed and all prior atoms are successfully executed, the last atom will deploy the two applications into the benchmark ByteDance Offline Environment. By clicking the atom, detailed execution output and logs can be found at the bottom.

To better manage pipelines, ByteCycle also supports the creation of pipelines using templates. Templates have components similar to pipelines including atoms with execution orders, variables, and triggers, but can be used to instantiate multiple pipelines. Project pipelines are often bound with templates, and when developers want to do some specific operations like deployment to production, they can choose the corresponding templates to generate a pipeline and complete the tasks. PIPELINEASCODE also provides the ability to manage templates in ByteCycle.

III. THE DESIGN OF PIPELINEASCODE

We design PIPELINEASCODE to receive a configuration file that describes all the information required for a pipeline as input and there are two main components of PIPELINEASCODE: the facade and the engine, as shown in Figure 2. PIPELINEASCODE is able to receive input configuration files from three resources: Software Development Kit (SDK), Command Line Interface (CLI), and Codebase at ByteDance. The SDK of PIPELINEASCODE allows users to call the service using their code, while the CLI provides a user interface where developers can input commands to manage pipelines of ByteCycle. PIPELINEASCODE can also connect to the codebase so that the pipelines can be started and the corresponding outcomes can be monitored when developers submit the code and trigger the codebase process on ByteDance. The facade component of PIPELINEASCODE is a web service that receives HTTP calls using a framework developed internally. The facade service assumes responsibility for processing requests originating from the three aforementioned resources while also guaranteeing the fulfillment of authentication prerequisites. The engine component of PIPELINEASCODE is a service that uses an RPC framework named Kitex [68]. It includes the core business logic of PIPELINEASCODE including the connection to the database and sending requests to ByteCycle. We will introduce the design of the PIPELINEASCODE engine in a more detailed way in the following subsections.

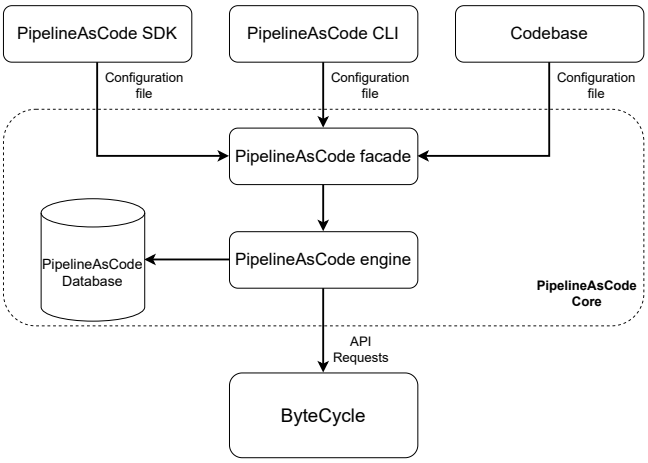

Fig. 2: Breif design of PIPELINEASCODE.

A. *Configuration File of* PIPELINEASCODE

The configuration file of PIPELINEASCODE is designed in Yaml format, while the fields are defined in protobuf [64]. The benefit of using protobuf is that it is validated and translated and is consistent with the coding habit of the US ByteCycle team. Each configuration file is assigned a distinct file identifier that has global uniqueness and is automatically generated through the PIPELINEASCODE service. Consequently, should developers aim to initiate a new pipeline, they are obliged to invoke the PIPELINEASCODE service to obtain a new file identifier. One configuration file is able to contain multiple pipelines, but only one of them can be the main pipeline, and other pipelines need to be sub-pipelines of the main pipeline. Moreover, it is worth mentioning that each pipeline is composed of multiple steps, wherein each individual step corresponds to a discrete atom. The determination of the appropriate atom to be utilized in a given step is contingent upon the specification provided in the "uses" field of that particular step. The step also includes the field of "dependsOn" to reflect the dependency relationship and the execution order of the steps: if step 1 depends on step 2, it means that step 1 needs to be executed after step 2 is completed. These pipelines are accompanied by additional information that serves to describe them, including triggers and the contents of variables. PIPELINEASCODE allows pipelines to be triggered on the basis of a git event or schedule. Lastly, another attribute, called the "workspace label", is used to indicate the specific workspace within which the pipeline will be instantiated.

An example of the configuration pipeline is shown in Listing 1. It is a simpler version of the pipeline introduced in Figure 1 that involves only the deployment of one software application rather than two software applications. From the example configuration file, we can find that the id is listed on the top and it only includes one pipeline. The pipeline, called "pac cd", encompasses a series of five distinct steps. The initial step initiates a shell and executes the specified commands to perform unit tests. Subsequently, the second step depends on the completion of the first step and proceeds to compile the software, taking into account the provided repository name and branch. Once the second step is finalized, the third step becomes viable, allowing the deployment of the package compiled from Step 2 into the testing environment. In Step 4, a separate shell is generated to perform integration testing. Finally, after the successful completion of all previous steps, the last step is triggered to deploy the compiled package from Step 2 to the benchmark environment. In addition, the trigger information is also listed in the configuration file: This specifies that this pipeline will be triggered when there are git merge events on the target branch of the target repository.

Listing 1: Configuration file example

```
id: p1b9a16a02eb
pipelines:
- id: pac_cd_4613549
  isMain: true
  name: pac cd
  steps:
  - dependsOn:
    - '-'
    id: devops_shell-7850e8
    input:
      run_context: |-
        go-byteview test -gcflags="all=-l -N"
            $(go list ./... | grep -v /dal)
        echo "byteview done"
      timeout: 1200
    name: Engine Unit Test
    uses: bytecycle/devops_shell
  - dependsOn:
    - devops_shell-7850e8
    id: scm_compile_beta-cca0f4
    input:
      scm_configs:
      - pub_base: branch_base
        revision: master
        scm_repo_name: canal/pac/engine
    name: SCM engine offline compilation
    uses: bytecycle/scm_compile
  - dependsOn:
    - scm_compile_beta-cca0f4
    id: boe_create_env-3d5178
    input:
      env_name: pac_staging
      host_type: docker
      keep_days: 15
      psm_list:
      - canal.pac.engine
    name: Deploy engine to BOE feature lane
    uses: bytecycle/boe_deploy
  - dependsOn:
    - boe_create_env-3d5178
    id: devops_shell-a99bcd
    input:
      run_context: |-
        go-byteview test ./integration_test
      timeout: 1200
    name: Integration Test
    uses: bytecycle/devops_shell
  - dependsOn:
    - devops_shell-a99bcd
    id: boe_create_env-9196c7
```

```
    input:
      psm_list:
      - canal.pac.engine
      - canal.pac.facade
    name: Upgrade facade and engine in BOE
        prod lane
    uses: bytecycle/boe_deploy
  triggers:
  - git:
      change:
        events:
        - GIT_MR_EVENT_MERGE
        repository: canal/pipeline_as_code
        targetBranch: master
workspaceLabel: bytecycle_pac
```

B. PIPELINEASCODE *Engine Service*

As the core service of PIPELINEASCODE, the engine service is an RPC service that functions as a mediator for processing requests originating from the facade service. It is responsible for translating these requests into the appropriate format, subsequently persisting them into a database, and invoking the ByteCycle APIs using a translated configuration file. The database we select for PIPELINEASCODE is an internally developed database based on MongoDB [3]. MongoDB is chosen as the preferred database solution due to its inherent flexibility and efficient querying capabilities. In particular, the absence of rigid relationships among files enables each file to be treated as an independent record within the database. Engine service offers a diverse set of functionalities to facilitate interaction with ByteCycle. Developers have the ability to create pipelines and subsequently initiate their execution. Moreover, developers can convert existing pipelines into configuration files and remove pipelines by specifying the corresponding file identifier. Furthermore, PIPELINEASCODE also provides some abilities for developers to manage templates, *e.g.,* create, delete templates, and bind the templates to projects. An additional crucial component within the engine service is the parser. This element plays an indispensable role in facilitating the conversion of configuration files to accommodate the different formats required for the PIPELINEASCODE engine, database and ByteCycle server.

IV. RESEARCH METHOD

In this section, we discuss how we collect the data set and what metrics we use to evaluate PIPELINEASCODE, and introduce the details of the evaluation process.

A. Data Set

We perform our study over the dataset which includes all pipelines managed by PIPELINEASCODE. The pipeline ids of the pipelines can be found in the database of PIPELINEASCODE in the field of "bc_pipeline_id". The database includes 2731 pieces of records, which means that 2731 configuration files have been created through PIPELINEASCODE. Out of the 2731 records we have, 1779 are pipeline configuration files and the rest are template configuration files. We did not include template configuration files in the evaluation experiments because templates cannot be executed and have a very similar structure to pipelines. Sometimes, more than one configuration file is used to manage the same pipeline. After we remove these duplicates, we are left with 452 different pipelines which are currently managed by PIPELINEASCODE. The initial data entry was generated on 20 October 2022. The selected temporal interval for this study was one year, indicating that all data was produced prior to October 20, 2023. We collect the information of pipeline id, build number, build status, create time, update time, and the step graph for each pipeline.

In an effort to understand the types of pipelines more inclined to utilize PIPELINEASCODE, we additionally collect data on pipelines not regulated by PIPELINEASCODE from ByteCycle. We executed a random selection of 452 pipelines (the same amount of pipelines as in the previous paragraph) that were instantiated within an identical time frame when the 452 pipelines managed by PIPELINEASCODE were formed. We also perform a filter process that excludes all pipelines that are only created and never triggered to build. The corresponding information from these non-PIPELINEASCODE-managed pipelines has been collected to be examined. We provide multiple statistics about the projects studied in Table I. Pipelines managed by PIPELINEASCODE have a total of 14546 builds, and all these builds are triggered between October 2022 and October 2023, while pipelines not managed by PIPELINEASCODE have a smaller number of builds (13167) and the builds have a similar time span of creation.

B. Metrics

To understand what pipelines adopt PIPELINEASCODE and how pipelines have changed after the adoption of PIPELINEASCODE, we measured five metrics in this evaluation: *Success_Rate*, *Build_Frequency*, *Change_Frequency*, *Build_Duration*, and *Step_Number*. We measured each of these metrics across all builds of a pipeline. Then we plot the result of each metric in a box plot, where each box represents the distribution of values for all the studied pipelines.

Success_Rate is the proportion of successful builds among all builds. This metric measures how reliable and efficient the pipeline is. It also reflects the benefit of CI to detect errors at an earlier stage. If a pipeline has a higher success rate after the adoption of PIPELINEASCODE, it means that PIPELINEASCODE may be able to reduce the failures of pipeline builds and improve the efficiency of artifact deployment.

Build_Frequency is measured as the daily frequency of triggered builds. This metric measures the active level of the pipeline and the ability of the pipeline to detect failures at an early stage. If a pipeline has a higher build frequency after the adoption of PIPELINEASCODE, it means that the adoption of PIPELINEASCODE may encourage developers to build more often, allowing them to find faults in advance. If pipelines with higher build frequency prefer to use PIPELINEASCODE, it may suggest that PIPELINEASCODE offers certain benefits that are particularly advantageous for pipelines where frequent updates, modifications, or other activities are taking place.

TABLE I: Characteristics of the studied pipelines.

Pipeline Type	# pipelines	# builds	Earliest Creation Time	Latest Creation Time
Pipelines managed by PIPELINEASCODE	452	14546	20-Oct-22	20-Oct-22
Pipelines not managed by PIPELINEASCODE	452	13167	23-Oct-23	18-Oct-23

Change_Frequency is measured as the proportion of builds where the pipeline is changed among all builds. For example, if one pipeline is triggered four times and its composition is changed only once in this period, the value of this metric will be 0.25. The frequency of pipeline modification represents an empirical measure of the flexibility of a pipeline. If pipelines characterized by a higher rate of changes prefer to use PIPELINEASCODE, it could indicate the advantages of PIPELINEASCODE to make changes easier in pipelines. This can also be confirmed if the pipeline is modified more frequently after the adoption of PIPELINEASCODE.

Build_Duration is measured as how long it takes to execute each build in seconds. It can partially reflect the complexity of the pipeline. Pipelines with longer duration may want to use PIPELINEASCODE because it can provide high reliability to these complex pipelines and ensure that it does not need to be rerun and takes extra time. If a pipeline takes a longer duration after adopting PIPELINEASCODE, it may reflect that developers tend to add more time-consuming atoms to the pipeline.

Step_Number is the number of steps that are included in the pipeline when the build is triggered. We use the median value of the step number across builds to represent the pipeline step number. This metric can be used to represent the complexity of the pipeline. If the pipeline tends to have more steps after the adoption of PIPELINEASCODE, then it may indicate that PIPELINEASCODE provides developers with more convenient ways to edit the pipelines to make it more complicated. It may also suggest that the PIPELINEASCODE configuration files are easy to understand and modify if pipelines with more steps prefer to adopt the tool.

C. Evaluation Process

To answer RQ1, we calculated the value of each feature for each pipeline and plot the result in a box plot. Each box includes all values for every studied pipeline. We compared the box generated from all pipelines that are managed by PIPELINEASCODE and all pipelines that have not adopted PIPELINEASCODE. We tested our results for statistical significance with a Wilcoxon rank-sum test (Mann-Whitney U test) because the samples are not paired, and decided statistical significance for $p < 0.05$.

To answer RQ2, we followed a similar process as RQ1, but only considered pipelines that adopt PIPELINEASCODE and split the builds of each pipeline into two parts: before the adoption of PIPELINEASCODE and after the adoption of PIPELINEASCODE based on the first build that was triggered by PIPELINEASCODE. We also drew box plots as the prior experiment, and each box has the values for all studied pipelines. In this experiment, the samples are related and paired because each data point pair represents the same pipeline in different time periods, so we used Wilcoxon signed-rank test for statistical significance test and also decided statistical significance for $p < 0.05$.

V. RESULTS

A. RQ1: What types of pipelines do developers prefer to manage using PIPELINEASCODE?

The results of RQ1 are plotted in Figure 3. This figure shows the median value for each metric in all pipelines studied. The Y axis is the evaluation metric, and each box contains the result of all pipelines. The X axis has the meaning of pipelines managed vs. not managed by PIPELINEASCODE. We make a few observations from our results. First, the pipelines managed by PIPELINEASCODE have a success rate with a median value of 40%, a build frequency with a median value of 0.94, a change frequency with a median value of 0.05, a build duration with a median value of 494 seconds and a step number with a median value of 7. This shows that these pipelines are generally rarely changed. Observing pipelines that have not been managed by PIPELINEASCODE, we can find that the data from these pipelines show a median success rate of 51%, a median build frequency of 0.34, a median change frequency of 0.02, a median build duration of 482.5 seconds and a median step count of 12. These observations show that ByteCycle pipelines are also rarely modified and a pipeline has a median value of 12 steps or atoms, making it relatively complex.

By comparing the characteristics of the pipelines managed and not managed by PIPELINEASCODE using the five metrics, we can observe that the pipelines managed by PIPELINEASCODE have a higher build frequency, a higher change frequency and a longer build duration, while the pipelines not managed by PIPELINEASCODE have more steps and a higher success rate. These differences were statistically significant ($p < 0.05$). We posit that developers may prefer to apply PIPELINEASCODE to those pipelines that are often required to build and change, are generally less complicated and are more likely to break. We think this may be because PIPELINEASCODE is capable of making pipeline modifications simply by editing the configuration files and PIPELINEASCODE can trigger the pipeline in a more convenient way because it accepts multiple input resources. However, those more complex pipelines are less likely to adopt PIPELINEASCODE because too many steps can make the configuration file too long to read and comprehend, especially compared to the visualization user interfaces.

B. RQ2: How do pipelines evolve after adopting PIPELINEASCODE?

Our analysis of RQ2 is presented in Figure 4, which contains the values for each metric of the pipelines studied.

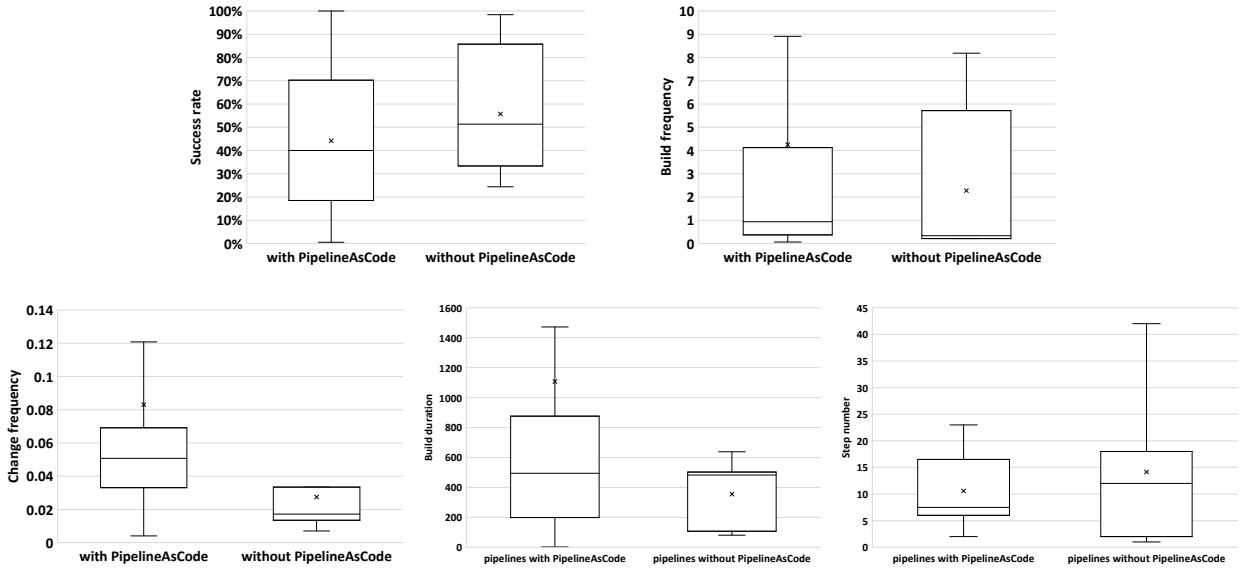

Fig. 3: Characteristics of pipelines with vs. without PIPELINEASCODE

The Y-axis represents the evaluation metric, and each box contains the data for all pipelines. The X-axis distinguishes the pipelines before the adoption of PIPELINEASCODE compared to the same pipelines after the adoption of PIPELINEASCODE. On the basis of the results, we can make a few observations. First, pipelines before adopting PIPELINEASCODE have a success rate with a median value of 20%, a build frequency with a median value of 0.83, a change frequency with a median value of 0.02, a build duration with a median value of 210 seconds and a step number with a median value of 7.5. After the adoption of PIPELINEASCODE, the median change frequency of pipelines increases to 0.66, the median build frequency increases to 0.95, the median build duration increases to 531 seconds, and the median success rate increases to 45%. These differences were statistically significant ($p < 0.05$). Therefore, we depict that the adoption of PIPELINEASCODE may be able to promote ByteCycle pipeline reliability by reducing the number of failed builds.

As the major benefit of continuous deployment lies mainly in the faster delivery of features, quality and customer satisfaction [50], PIPELINEASCODE can maximize the benefit because it improves the reliability of the pipelines so that the pipelines can deliver quality software to end users on time. Furthermore, we also find that adopting PIPELINEASCODE allows pipelines to be triggered more frequently. This is also beneficial because more frequent builds obey the best practice of Continuous Integration and Deployment. Thanks to the increasing build frequency, CI/CD users can identify errors at an earlier stage, preventing them from becoming too significant to resolve. Additionally, we can also observe that the frequency of change in pipelines also increases with the adoption of PIPELINEASCODE. This may reflect that PIPELINEASCODE can promote the flexibility of the CI/CD pipelines because users are more willing to adjust the pipelines based on demands or preferences. The increasing flexibility of pipelines can ensure that the pipeline is up-to-date and performs testing and deployment on the basis of the latest needs. In addition, the duration of pipelines has also been increasing since the adoption of PIPELINEASCODE. This may be because users recently changed some steps that are more time-consuming or because the builds are more frequent, which requires CI/CD users to do some manual operations on the pipelines more often. Users may want to explore the reason for the longer duration to maintain the high efficiency of the pipelines. The last trend that we observe in Figure 4 is that pipelines tend to have fewer steps after the adoption of PIPELINEASCODE. This might suggest that CI/CD users may find it less convenient to use configuration files to manage pipelines with a long list of steps because it is hard to clearly understand the execution order of the steps without visualization. This may also imply that one main bottleneck of applying the idea of "as code" practice is that it requires more knowledge transfer and the configuration files are less accessible compared to user interfaces.

VI. DISCUSSION

In this section, we discuss the challenges encountered and lessons learned during the process of designing PIPELINEASCODE and achieving its standards to meet user requirements. We also discuss the forthcoming challenges that need to be addressed in the system.

A. Lesson Learned

The first thing we learned about is the trade-off between using configuration files and the user interface when describing CI/CD pipelines. The trade-off lies in the aspects of flexibility, transparency, history tracking, and ease of use. The user interface is generally more intuitive and user-friendly. It is able to provide a visualization for each pipeline, and the

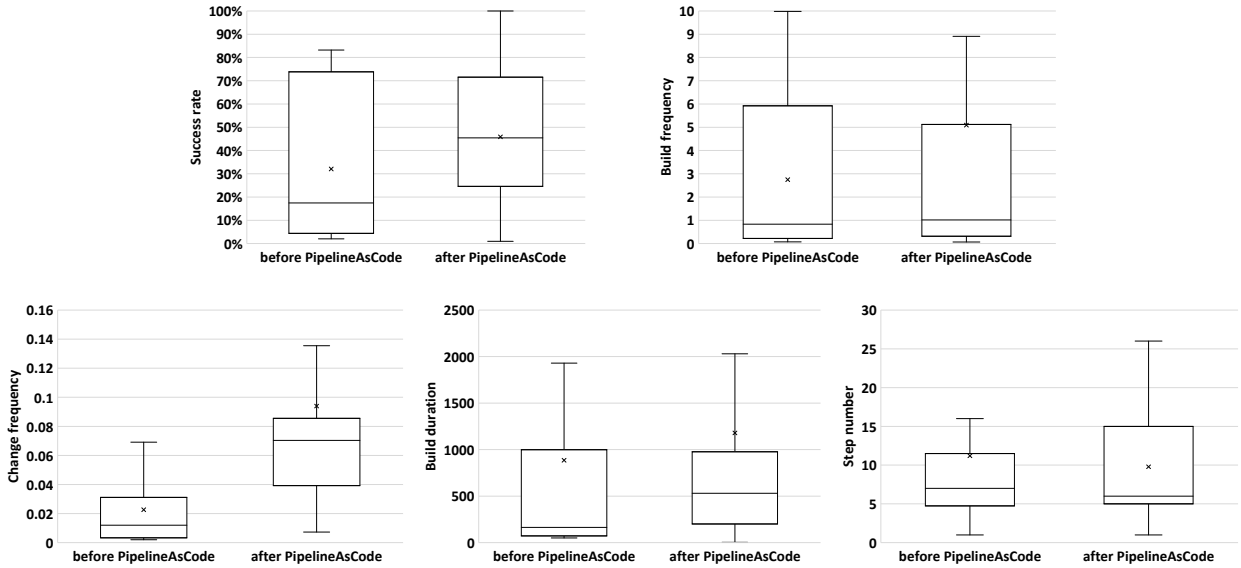

Fig. 4: Characteristics of pipelines before vs. after adopting PIPELINEASCODE

changes will have an immediate impact. Users can clearly understand the structure of the pipeline and easily query the status. However, user interfaces also limit the ability of porting pipelines, and it is hard to track the change history and realize the version control on user interfaces. On the other hand, configuration files enable easier pipeline replication and it is convenient to win the version control by checking the file into codebase repository. Configuration files can also provide users with a better flexibility to make modifications to the pipelines. These were the motivations of PIPELINEASCODE. However, there are also drawbacks of using configuration files because they are difficult to understand without background knowledge and difficult to find the error when editing the file, resulting in a trade-off between user interfaces and "as code" practices.

Another lesson we learned during the design of PIPELINEASCODE is how to overcome the security issue. PIPELINEASCODE faces both the issues of authorization and authentication: the system must identify the input creator and also decide what permission the creator should have. To address this, the PIPELINEASCODE command line interface takes advantage of ByteDance Single Sign On system that requires users to log in through QR code when using the tool, and their information will be cached for a specific period of time. When sending requests to the PIPELINEASCODE service, the user's JSON Web Token will be packaged as well for the authorization requirement from ByteCycle. Other lessons we learned include the input diversity requirement, the trade-off between Protobuf and JSON, and the appropriate selection of databases.

B. Forthcoming Challenges

One challenge that must be overcome is how to detect errors when editing configuration files. As mentioned in §VI-A, it is hard to find the error when editing the configuration file. Currently, the integrated development environment (IDE) cannot detect errors when modifying the PIPELINEASCODE configuration files, as the user interface can. Therefore, future plugins can be developed to support the detection of typos and incorrect input in the configuration files. This also raised another challenge to locate the bug in the configuration file and the automatic fix suggestion given the log of a broken build. Although the existing work [25] aims to repair the build scripts automatically, the evaluation results still suggest that these tools can only fix a small proportion of failures with limited accuracy and have not been used in industrial scenarios.

Another forthcoming challenge that requires to be solved is how to simplify the configuration files as much as possible so that it saves users' effort when doing knowledge transfer. Our PIPELINEASCODE is designed to simplify the input of those steps by removing the unnecessary fields, however, those inputs still look very complicated and this has greatly restricted the adoption of PIPELINEASCODE. From §V, we can also find that pipelines with less complex structures are more likely to use PIPELINEASCODE for management, and after users become familiar with the tool, the number of steps in the pipelines starts to increase. Therefore, the development of tools that are able to shorten the input length of the steps or auto-fill the configurations is motivated. Another challenge is related to how to keep the uniqueness of the source of truth as it can come from multiple resources such as configuration files and user interface. Other challenges to be addressed include strong dependency issues of PIPELINEASCODE on other platforms, making it difficult to be resolved to other places.

VII. THREATS TO VALIDITY

A. Internal Validity

To guard internal validity, we collected all pipelines managed by PIPELINEASCODE and all pipelines have at least one build. Our analysis could also be influenced by uncompleted

information in our analyzed dataset because we randomly select a subset of pipelines in the ByteCycle database. For this, we guaranteed a random selection process and performed a filter process to exclude all pipelines that have no build history. To ensure that we compare things accurately and fairly over time, we kept the same number of pipelines managed by PIPELINEASCODE as those not managed by it. We also made sure that these pipelines were created at the same time as when we started using PIPELINEASCODE. This ensures that the data is up-to-date.

The analysis of the success rate may also influence the proportion of builds whose status is "pending". The "pending" status indicates that the build is currently being held and requires some manual confirmation from the users. These builds may eventually become successful if users operate correctly on them. We did not count builds with such a status as successful builds because the pending duration in the collected data set has exceeded one day, and thus we consider these builds broken. Our results may also be affected by flaky tests that cause spurious failure builds. However, CI/CD systems are expected to function even in the presence of flaky tests, since most companies do not consider it economically viable to remove them *e.g.,* [43], [46]. Our observed pipeline runtimes may have been influenced by the load experienced in the build server at the time. However, we consider this potential impact to be very low and it may influence all pipelines, regardless of whether they have or have not adopted PIPELINEASCODE. Another factor that may influence the findings is the logical inferences and conclusions drawn from the research results. However, our evaluation process controlled the variables by comparing the same pipeline before vs. after the adoption of PIPELINEASCODE and the results are statistically significant.

B. External Validity

To increase external validity, we selected all pipelines that were included in the PIPELINEASCODE database and randomly collected the same number of pipelines from the ByteCycle database after a filtering process. The pipelines we chose were mostly using Golang because Golang is the dominant programming language used at ByteDance. Although this programming language is widely used, different CI habits in other languages may provide slightly different results from those in this study. Our observations may slightly vary for separate software projects, but our goal was to derive general observations for a real-world population of software projects. Given that the pipelines we collected were all from industrial areas, the results may not be always applicable to some smaller open-source projects.

C. Construct Validity

A threat to the validity of the construct is whether we studied software projects that are similar to those in the real world. However, we randomly selected the pipelines from the database, and all the projects selected were industrial projects, which can ensure that they have similar characteristics to other real-world projects. Another threat to construct validity is whether the metrics measured are able to reflect the reliability and flexibility of CI/CD pipelines. However, existing work [6], [28] has determined that the benefit of the CI / CD process includes early failure detection and a faster deployment pace and that the failure of the builds can negatively influence the reliability of the process negatively. In this paper, we also use the change frequency to represent the flexibility of the pipelines because more frequent modifications on the pipeline can ensure that the pipeline meets the latest requirements of the projects and gets adjusted timely. We also used the build duration and the number of steps to represent the complexity of the pipeline, as many existing works did [19], [24], [32].

To answer RQ1, each data point in the box plot is from different pipelines and is not paired with each other. Therefore, we used the Wilcoxon rank-sum test (Mann-Whitney U test) for the statistical significance test. On the other hand, each data point in the box plot to answer RQ2 is paired with each other because they refer to the characteristics of the same pipelines before and after the adoption of PIPELINEASCODE. Thus, we selected the Wilcoxon signed-rank test as the method of the statistical significance test.

VIII. RELATED WORK

A. Empirical Studies of CI/CD with Cost and Benefit

Multiple researchers focused on understanding the practice of CI, studying both practitioners *e.g.,* [28] and software repositories [66]. Vasilescu *et al.* studied CI as a social coding tool [65], and later studied its impact on software quality and productivity [66]. Zhao *et al.* studied the impact of CI in other development practices, such as bug fixing and testing [74]. Stahl *et al.* [62] and Hilton *et al.* [28] studied the benefits and costs of using CI and the trade-offs between them [27]. Lepannen *et al.* studied the costs and benefits of continuous delivery in a similar way [40]. Felidré *et al.* [14] studied the adherence of projects to the original CI rules [15]. Other recent studies analyzed testing practices [18], difficulties [51], and pain points [69] in CI. The high cost of running builds is highlighted by many empirical studies as an important problem in CI [26], [28], [27], [51], [69]. People [27], [66] believe that the benefit of CI lies mainly in early fault detection. Others [28], [40] find that projects adopting CI are able to adopt pull requests and release in a shorter time. Some also find that CI can help the developer team in other areas, such as providing a common building environment [27] and increasing team communication [62].

Other researchers worked to explain why CDE is adopted and report the huge benefits and challenges involved [6]. Chen *et al.* presented strategies to help overcome the challenges of CDE [7] and Virmani *et al.* also explained how to bridge the gap of CDE to promote the adoption [67]. Savor *et al.* explained how CD has been achieved at Facebook [58]. Existing work [8] also examined the challenges faced by organizations when adopting CD, as well as strategies to mitigate these challenges. Yang *et al.* explored two types of workflow in CD including Docker Hub auto-builds Workflows and CI-based Workflows [73]. Other works also explained the

difference between CD and CDE and their own approaches and challenges [59], [60].

B. Approaches to Improve CI/CD

A related effort to improve CI aims at speeding up its feedback by prioritizing its tasks. The most common approach in this direction is to apply test case prioritization (TCP) techniques *e.g.,* [12], [13], [42], [44], [49], [57], [76] so that builds fail faster. These techniques, although not designed to work in a CI environment, have been claimed to have the potential to provide CI users with earlier fault observation. Another similar approach achieves faster feedback by prioritizing builds instead of tests [41]. A popular effort to reduce the cost of CI focuses on understanding what causes long build durations *e.g.,* [19], [63]. Thus, most of the approaches that reduce the cost of CI aim at making builds faster by running fewer test cases on each build. It is found that a lot of passing tests could be saved in this way [39].

Some approaches use historical test failures to select tests [13], [26]. Others run tests with a small distance to code changes [45] or skip testing unchanged modules [61]. Recently, Machalica *et al.* predicted test case failures using a machine learning classifier [43]. These techniques are based on the broader field of regression test selection (RTS) *e.g.,* [21], [55], [56], [70], [71], [72], [75]. While these techniques focus on making every build cheaper, other work addresses the cost of CI differently: by reducing the total number of builds that get executed. A related recent technique saves cost in CI by not building when builds only include non-code changes [1], [2]. They first create a rule-based selection technique and then take advantage of the machine learning algorithm to improve the accuracy. Then Jin and Servant propose multiple build selection tools to reduce computational cost [29], [30], [32], [35], [36]. They also design an evaluation across all CI-improving techniques to compare their own benefits [33], [34].

Finally, other complementary efforts to reduce build duration have targeted speeding up the compilation process *e.g.,* [5] or the initiation of testing machines *e.g.,* [17]. Regarding improving CD, Gallaba *et al.* explore ways to improve the robustness and efficiency of CD processes [16]. Also, existing work aims to aid product teams in improving their deployment process through characterizing experimentation in CD [37]. Our work targets improving CI/CD workflows by improving their flexibility and reliability so that CI/CD users are able to build more often to detect failures earlier and deploy more often to release their applications.

C. Infrastructure as Code

Infrastructure as Code (IaC) has been popular as automation technologies [48]. Four topics studied in IaC-related publications are identified by existing work [53]: (i) framework/tool for infrastructure as code; (ii) use of infrastructure as code; (iii) empirical study related to infrastructure as code; and (iv) testing in infrastructure as code. Guerriero *et al.* explore the adoption, support and challenges of IaC and highlight the need for more research in the field: The support provided by currently available tools is still limited, and developers feel the need for novel techniques to test and maintain IaC code [23]. Other existing work focuses on helping practitioners improve the quality of infrastructure as code (IaC) scripts by identifying development activities related to defective IaC scripts [52] and avoiding insecure coding practices while developing IaC scripts through an empirical study of security smells in IaC scripts [54].

As another "as code" practice similar to Infrastructure as Code, pipeline as code is introduced as a challenge for many software engineering teams as it requires the use of many tools and processes that all work together [38]. Existing work introduces how a pipeline is written outside the Jenkins UI in a file which refers to the process of pipeline as code and the advantages of it including creating multiple pipelines using one configuration file, conducting code review on the configuration file and using version control system to keep track of its change history [10]. Our PIPELINEASCODE tool is designed based on the unique context of developing environment at ByteDance and it also has some differences between what is done in Jenkins: we allow freestyle atoms from public atom markets and implement a more strict rule that each configuration can only be bound with one main pipeline. We also propose the first evaluation on pipeline as code tools to reflect its benefits in improving the flexibility and reliability of CI/CD pipeline system.

IX. CONCLUSIONS AND FUTURE WORK

In this article, we introduced how Continuous Integration and Continuous Deployment pipelines work on the ByteCycle platform at ByteDance. We also introduced our design of PIPELINEASCODE, a CI/CD workflow management system through configuration files. We performed two evaluation experiments on PIPELINEASCODE to understand the characteristics of the pipelines to adopt them and their influence on the pipelines. We compared the results of the pipelines with vs. without PIPELINEASCODE and the pipelines before vs. after adopting PIPELINEASCODE on five metrics in both experiments. We derived many observations from the evaluation, which we then synthesized to understand the benefits and potential limitations of adopting PIPELINEASCODE on the CI / CD pipelines. We observed that PIPELINEASCODE is capable of improving the reliability and flexibility of pipelines and increasing the build frequency to maximize the ability to detect failures at an early stage. Our findings can shed light on the design of future tools. Finally, we discuss what we had learned during the design of PIPELINEASCODE and the gap between the initial design and how developers would like to use the tool. We also provided a set of challenges or pain points that may require future work to address. We lay out plans to simplify the construction of the configuration files to enhance their flexibility and practicality. In the future, we will work on smart pipeline generation, auto-code completion of the configuration files (such optimization decisions may have hidden costs [31]), and the fault localization of build failures caused by the incorrect input of the configuration files.

REFERENCES

[1] R. Abdalkareem, S. Mujahid, and E. Shihab. A machine learning approach to improve the detection of ci skip commits. *IEEE Transactions on Software Engineering (TSE)*, 2020.

[2] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling. Which commits can be ci skipped? *IEEE Transactions on Software Engineering*, 2019.

[3] K. Banker, D. Garrett, P. Bakkum, and S. Verch. *MongoDB in action: covers MongoDB version 3.0*. Simon and Schuster, 2016.

[4] M. Beller, G. Gousios, and A. Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 356–367. IEEE, 2017.

[5] A. Celik, A. Knaust, A. Milicevic, and M. Gligoric. Build system with lazy retrieval for java projects. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–654. ACM, 2016.

[6] L. Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE software*, 32(2):50–54, 2015.

[7] L. Chen. Continuous delivery: overcoming adoption challenges. *Journal of Systems and Software*, 128:72–86, 2017.

[8] G. G. Claps, R. B. Svensson, and A. Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software technology*, 57:21–31, 2015.

[9] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*, pages 1277–1286, 2012.

[10] P. Dingare. Understanding pipeline as code. In *CI/CD Pipeline Using Jenkins Unleashed: Solutions While Setting Up CI/CD Processes*, pages 307–338. Springer, 2022.

[11] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.

[12] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering*, 28(2):159–182, 2002.

[13] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–245, 2014.

[14] W. Felidré, L. Furtado, D. A. Da Costa, B. Cartaxo, and G. Pinto. Continuous integration theater. In *Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 10, 2019.

[15] M. Fowler and M. Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122:14, 2006.

[16] K. Gallaba. Improving the robustness and efficiency of continuous integration and deployment. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 619–623. IEEE, 2019.

[17] A. Gambi, Z. Rostyslav, and S. Dustdar. Improving cloud-based continuous integration environments. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 797–798. IEEE Press, 2015.

[18] A. Gautam, S. Vishwasrao, and F. Servant. An empirical study of activity, popularity, size, testing, and stability in continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 495–498. IEEE, 2017.

[19] T. A. Ghaleb, D. A. da Costa, and Y. Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, pages 1–38, 2019.

[20] GitLab. Ci cd. https://about.gitlab.com/topics/ci-cd/, 2023. [Online; accessed 29-October-2023].

[21] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[22] M. Golzadeh, A. Decan, and T. Mens. On the rise and fall of ci services in github, Oct. 2021.

[23] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *2019 IEEE international conference on software maintenance and evolution (ICSME)*, pages 580–589. IEEE, 2019.

[24] F. Hassan and X. Wang. Change-aware build prediction model for stall avoidance in continuous integration. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 157–162. IEEE, 2017.

[25] F. Hassan and X. Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1078–1089. IEEE, 2018.

[26] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 483–493. IEEE, 2015.

[27] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 197–207. ACM, 2017.

[28] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 426–437. ACM, 2016.

[29] X. Jin. Reducing cost in continuous integration with a collection of build selection approaches. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2021.

[30] X. Jin. *Cost-saving in Continuous Integration: Development, Improvement, and Evaluation of Build Selection Approaches*. PhD thesis, Virginia Tech, 2022.

[31] X. Jin and F. Servant. The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency. In *Proceedings of the 15th International conference on mining software repositories*, pages 70–73, 2018.

[32] X. Jin and F. Servant. A cost-efficient approach to building in continuous integration. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 13–25. IEEE, 2020.

[33] X. Jin and F. Servant. Cibench: a dataset and collection of techniques for build and test selection and prioritization in continuous integration. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 166–167. IEEE, 2021.

[34] X. Jin and F. Servant. What helped, and what did not? an evaluation of the strategies to improve continuous integration. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 213–225. IEEE, 2021.

[35] X. Jin and F. Servant. Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration. *Journal of Systems and Software*, 188:111292, 2022.

[36] X. Jin and F. Servant. Hybridcisave: A combined build and test selection approach in continuous integration. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–39, 2023.

[37] K. Kevic, B. Murphy, L. Williams, and J. Beckmann. Characterizing experimentation in continuous deployment: a case study on bing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 123–132. IEEE, 2017.

[38] M. Labouardy. *Pipeline as code: continuous delivery with Jenkins, Kubernetes, and terraform*. Simon and Schuster, 2021.

[39] A. Labuschagne, L. Inozemtseva, and R. Holmes. Measuring the cost of regression testing in practice: a study of java projects using continuous integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 821–830, 2017.

[40] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *Ieee software*, 32(2):64–72, 2015.

[41] J. Liang, S. Elbaum, and G. Rothermel. Redefining prioritization: continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering*, pages 688–698, 2018.

[42] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta. Assessing test case prioritization on real faults and mutants. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 240–251. IEEE, 2018.

[43] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software*

Engineering: Software Engineering in Practice (ICSE-SEIP), pages 91–100. IEEE, 2019.

[44] D. Marijan, A. Gotlieb, and S. Sen. Test case prioritization for continuous regression testing: An industrial case study. In *2013 IEEE International Conference on Software Maintenance*, pages 540–543. IEEE, 2013.

[45] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242. IEEE, 2017.

[46] J. Micco. The state of continuous integration testing@ google. 2017.

[47] K. Morris. *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.

[48] K. Morris. *Infrastructure as code*. O'Reilly Media, 2020.

[49] S. Mostafa, X. Wang, and T. Xie. Perfranker: prioritization of performance regression tests for collection-intensive software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 23–34, 2017.

[50] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, et al. The top 10 adages in continuous deployment. *IEEE Software*, 34(3):86–95, 2017.

[51] G. Pinto, M. Rebouças, and F. Castor. Inadequate testing, time pressure, and (over) confidence: a tale of continuous integration users. In *Proceedings of the 10th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 74–77. IEEE Press, 2017.

[52] A. Rahman, E. Farhana, and L. Williams. The 'as code'activities: development anti-patterns for infrastructure as code. *Empirical Software Engineering*, 25:3430–3467, 2020.

[53] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams. A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108:65–77, 2019.

[54] A. Rahman, C. Parnin, and L. Williams. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175. IEEE, 2019.

[55] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, 22(8):529–551, 1996.

[56] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[57] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[58] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. Continuous deployment at facebook and oanda. In *Proceedings of the 38th International Conference on software engineering companion*, pages 21–30, 2016.

[59] M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu. Beyond continuous delivery: an empirical investigation of continuous deployment challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 111–120. IEEE, 2017.

[60] M. Shahin, M. A. Babar, and L. Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access*, 5:3909–3943, 2017.

[61] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka. Optimizing test placement for module-level regression testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 689–699. IEEE, 2017.

[62] D. Ståhl and J. Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th iasted international conference on software engineering,(innsbruck, austria, 2013)*, pages 736–743, 2013.

[63] M. Tufano, H. Sajnani, and K. Herzig. Towards predicting the impact of software changes on building activities. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, ICSE '19, 2019.

[64] K. Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 72:23, 2008.

[65] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. van den Brand. Continuous integration in a social-coding world: Empirical evidence from github. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 401–405. IEEE, 2014.

[66] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.

[67] M. Virmani. Understanding devops & bridging the gap from continuous integration to continuous delivery. In *Fifth international conference on the innovative computing technology (intech 2015)*, pages 78–82. IEEE, 2015.

[68] Y. Wen, G. Cheng, S. Deng, and J. Yin. Characterizing and synthesizing the workflow structure of microservices in bytedance cloud. *Journal of Software: Evolution and Process*, 34(8):e2467, 2022.

[69] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu. A conceptual replication of continuous integration pain points in the context of travis ci. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 647–658. ACM, 2019.

[70] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150. ACM, 2007.

[71] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[72] L. Zhang. Hybrid regression test selection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 199–209. IEEE, 2018.

[73] Y. Zhang, B. Vasilescu, H. Wang, and V. Filkov. One size does not fit all: an empirical study of containerized continuous deployment workflows. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 295–306, 2018.

[74] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 60–71. IEEE Press, 2017.

[75] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric. A framework for checking regression test selection tools. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 430–441. IEEE, 2019.

[76] Y. Zhu, E. Shihab, and P. C. Rigby. Test re-prioritization in continuous testing environments. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 69–79. IEEE, 2018.