Learning CI Configuration Correctness for Early Build Feedback

Mark Santolucito
Barnard College, Columbia University
NYC, USA
msantolu@barnard.edu

Jialu Zhang
Yale University
New Haven, USA
jialu.zhang@yale.edu

Ennan Zhai
Alibaba Group
Seattle, USA
ennan.zhai@alibaba-inc.com

Jürgen Cito
TU Wien
Vienna, Austria
juergen.cito@tuwien.ac.at

Ruzica Piskac
Yale University
New Haven, USA
ruzica.piskac@yale.edu

2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-3786-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SANER53432.2022.00118

***Abstract*—Continuous Integration (CI) allows developers to check whether their code can build successfully and pass tests across various system environments with every commit. To use a CI platform, a developer must provide configuration files within a code repository to specify build conditions. Incorrect configuration settings lead to CI build failures, which can take hours to run, wasting valuable developer time and delaying product release dates. Debugging CI configurations is a slow and error-prone process. The only way to check the correctness of CI configurations is to push a commit and wait for the build result. We present VeriCI, the first system for localizing CI configuration errors at the code level. VeriCI runs as a static analysis tool, before the developer sends the build request to the CI server. Our key insight is that the commit history and the corresponding build histories available in CI environments can be used both for build error prediction and build error localization. We leverage the build history as a labeled dataset to automatically derive customized rules describing correct CI configurations, using supervised machine learning techniques. To more accurately identify root causes, we train a neural network that filters out constraints that are less likely to be connected to the root cause of build failure. We evaluate VeriCI on real world data from GitHub and achieve 91% accuracy of predicting a build failure and correctly identify the root cause in 75% of cases. We also conducted a between-subjects user study with 20 software developers, showing that VeriCI significantly helps users in identifying and fixing errors in CI.**

***Index Terms*—configuration files, program analysis, continuous integration**

I. Introduction

Continuous Integration (CI) is seeing broad adoption with the increasing popularity of the GitHub pull-based development model [1]. There are now a plethora of open-source, GitHub-compatible, cloud-based CI tools, such as TravisCI [2], CircleCI [3], Jenkins [4], GitLab CI [5], Codefresh [6] and TeamCity [7]. Over 930,000 open-source projects are using TravisCI alone [2]. A CI platform provides developers with continuous feedback on every commit indicating if their code successfully built and whether the given tests passed. As such, CI has become an integral part of DevOps and the software development process more generally.

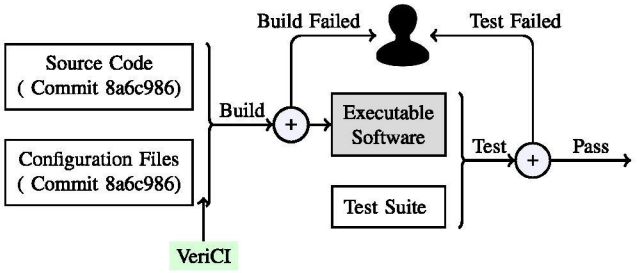

Fig. 1: A typical build process of CI. Our tool, VeriCI, statically analyzes a commit to predict failing builds and reports the root cause of the failure.

Continuous integration is an essential part of modern software development. In Fig. 1, we show the typical development workflow when using CI. Programmers develop code and write associated test suites. To build the code and run the test suite, each CI repository contains configuration files defining the execution steps. These configuration files specify build conditions such as the operating system, disk size, which compiler flags to use, which library dependencies are required, and other similar properties. Then, each time a developer commits new code, the CI platform builds multiple versions of the executable software according to the configuration, with each version corresponding to the specified build conditions. The CI platform then runs these executables on the test suite with appropriate hardware.

However, for all the benefits CI brings, it also suffers from some serious limitations. If the executable fails to build, the tests cannot be executed, and the developer must try to fix the configuration before the code can be checked for functional correctness. This can delay development significantly as CI builds regularly take on the scale of hours [8], [9]. Furthermore, if the CI configuration is incorrect (e.g. uses a library incompatible with the specified OS) and the project fails to build, the user does not receive any feedback on the

978-1-6654-3786-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SANER53432.2022.00118

functional correctness of their code. By running a query[1] on the historical build data available on TravisTorrent [10], we find that 662,442 hours or 75 years of server time was spent on builds that eventually errored. Such incorrect CI configurations result not only in lost server time, but also lost developer time, as developers must wait to verify that their work does not break the build. As one example, the release of a new version of the open-source project, SuperCollider, was delayed for three days because of slow TravisCI builds [11].

One of the main sources of difficulty in debugging CI configurations is that a developer cannot test the configuration for platforms that are not available locally. The only way to test a configuration is to push a commit and wait for the CI platform to complete the build. If these incorrect configurations could be quickly and statically checked on the client side before the build, the CI workflow would be considerably improved.

CI Configurations Analysis. To address these issues, we developed a tool, VeriCI, for identifying potential errors in a given CI configuration *before* the developer sends the build request to the CI platform. VeriCI employs a language agnostic approach that learns correctness criteria for CI environments by extracting features we call *magic constants*. The magic constant features embodies the use of hard-coded numerical constants in a repository (e.g., library version constants, see Sec. IV-A), as they often represent code locations that are prone to misconfigurations prevalent in CI. VeriCI works in two stages: first, it predicts build failures based on code analysis of the incoming commit (based on our extracted features), and second, if the build is predicted to error, VeriCI generates an error message that localized the root cause of the error in the code.

Build prediction has been extensively studied in the software engineering community [12], [13], [14], [15], [16], [17], [18]. However, these approaches largely rely on metadata of the repository and commits for prediction (commit messages, commit size, repository size, etc.). Since the prediction relies on metadata, the prediction cannot be deconstructed to localize the error to a specific line of code. This is a key restriction on prior work which can only provide *file level* fault localization [16], [17], not *code level* fault localization. To precisely localize build failures in the code base, our tool is based solely on analysis of code and does not utilize metadata associated with the repository.

While there has also been prior work on configuration error localization that does not utilize code metadata [19], [20], [21], [22], it cannot be applied in this setting as these techniques are limited to traditional key-value configuration languages.

In VeriCI, we leverage two unique properties of the CI environment. First, as a result of integration with version control systems, CI also comes with a dataset of build logs over the commit history. This dataset is inherently a labeled set, where the commits are classified as passed or failed. We use the commit history as our training set, and use supervised learning techniques, such as decision trees and neural networks, to learn a specification of the CI behavior. Using this specification (as a set of constraints on configuration values), we then check future commits for potential build errors. VeriCI continually evolves and refines its learned model as more commits enter the repository history. One key component of the supervised learning approaches we use is the selection of features. We base our feature extraction on hard coded numerical constants in code, with the intuition that these are likely to encode configurations settings across the code base. The use of numerical constants as features allows us to learn rules such as version inconsistencies between libraries.

The second unique property we gain from the CI environment is that in our training set, as a product of being a version controlled repository, there is an incremental change between successive commits. This allows us to narrow down the search space for the root cause of the failed builds. To detect these failed builds, we leverage decision tree learning, specialized to the task of generating error messages that help users identify the root cause of a build failure. We additionally combine the decision tree algorithm with a neural network to increase the accuracy of our error localization.

We evaluated VeriCI on real world data from GitHub. VeriCI achieves 91% accuracy when predicting a build failure and correctly identifies the root cause in 75% of cases. Additionally, to assess the efficacy of generated error messages, we conducted a between-subjects user study with 20 software developers. The control group had access to the standard TravisCI output, while the treatment group was additionally provided with the output of VeriCI. Our study shows that VeriCI significantly helps users in identifying and fixing errors in CI. However, in cases where the root cause is easily extractable from the log file, we did not see significant differences between control and treatment groups.

In summary, this work makes the following contributions:

1) Introduces the first approach to predicting CI build status that also identifies specific lines of code as probable root cause locations. Our approach introduces Abstraction Based Relabelling (ABR), a data set relabelling technique that mitigates the potential of predicting false positives from noisy CI build data.
2) Promotes the use of a restricted set of features during the learning process as a way to achieve classification performance on par with existing tools, while additionally localizing potential errors.
3) An evaluation of our tool, VeriCI, that achieves 91% accuracy of predicting a build failure and identifies the root cause location in 75% of cases.
4) A user study that shows developers using VeriCI can identify and fix potential CI build errors more accurately for some cases, while highlighting limitations for others.

II. Motivating Examples

We give here two examples to illustrate the types of failures that developers face in the CI environment. In these examples,

[1]https://console.cloud.google.com/bigquery?sq=737774871812:615ddd94dc2a40e592ea8fca8072a355

VeriCI generates error messages to guide developers to identify the root cause of the build failure. All of the following examples are taken from the existing repositories and they illustrate real issues that have occurred during the build process.

A. Identifying an error within a single file

We start with an example of a TravisCI build failure from the sferik/rails_admin repository [23] on GitHub. This is a large repository with (at the time of writing) 4603 commits over 403 contributors, 7400 stars, and 2100 forks.

In commit `3fd3b32` an administrator merged the changes from a pull request into the master branch, which caused the TravisCI build to fail. The pull request changed 3 files by adding 125 lines and deleting 10 lines. Manually checking the log information is a tedious process that, in this case, did not provide any helpful information for understanding or correcting the issue. As a result, the next 14 commits in the repository still have the same failed TravisCI status. It was not until 20 days later that a contributor to the repository was able to correct this build failure [24].

We ran VeriCI on commit `3fd3b32`, which caused the build to fail, and VeriCI correctly predicted a build failure. Additionally, VeriCI also provided an explanation for its classification, by reporting two substrings from the codebase which are likely to be contributing to the build failure.

```
Predicted build failure based on these keywords
   - if options[:encoding_to].present? &&
        @encoding_to == Encoding::UTF_
   - rvm
```

We again confirmed this as a true positive, as this is exactly the location of the change made by the user when fixing the build failure [24]. VeriCI identified the "minimal" problem, i.e., the smallest set of commands that the build system was not able to handle. Since there may be multiple solutions to a build failure, VeriCI does not try to suggest a repair, but rather helps the user identify the root cause of a build failure. In this way, users can prioritize the potential problems using their knowledge of the code base.

B. Identifying errors spanning multiple files

In the previous example, VeriCI detected the root cause of an error which required changing a single line. However, many CI errors can be a result of complex relationships between multiple files and branches in a repository. To demonstrate this, we take another real-world example, from the activescaffold/active_scaffold [25] repository which has (at the time of writing) 5326 commits over 92 contributors, 998 stars, and 327 forks.

Fig. 2 shows a sequence of commits. Each commit is represented by a box and the arrows point from a parent commit to a child commit. A commit can have two child commits (as in commit `a086ca7`) when two users make a different change based on the same state of code. This difference was eventually resolved through a merge, resulting in a commit that has two parent commits (as in the case of commit `bce9420`).

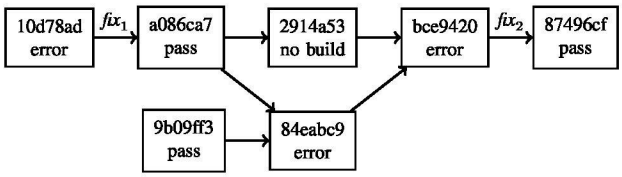

Fig. 2: A GitHub history chain depicting dependencies between commits.

To better understand these commits, the `git diff` command shows the difference between the current and previous commit. Fig. 3 depicts the `git diff` from a selection of commits from Fig. 2. Italics indicate which file was changed, while (+/-) indicate what was added or removed.

Listing 1: 10d78ad

version.rb

```
module Version
 MAJOR = 3
 MINOR = 4
- PATCH = 34
+ PATCH = 35
```

Listing 2: a086ca7

Gemfile.lock

```
- active_scaffold
     (3.4.34)
+ active_scaffold
     (3.4.35)
```

Listing 3: bce9420

```
- Gemfile.lock
+ Gemfile.rails – 4.0.x.lock
+ Gemfile.rails – 4.1.x.lock
+ active_scaffold
     (3.4.34)
```

Listing 4: 87496cf

Gemfile.rails – 4.0.x.lock, *Gemfile.rails – 4.1.x.lock*

```
- active_scaffold
     (3.4.34)
+ active_scaffold
     (3.4.35)
```

Fig. 3: The git diffs for a selection of the commits from Fig. 2 illustrating how our feature extraction process (*magic constants*) can provide a proper abstraction for identifying failures based on configurations in CI

In commit `10d78ad` in Fig. 2, the author upgraded the version number of the package from 3.4.34 to 3.4.35 in the `version.rb` file [25]. However, in order to correctly bump the version number in this project, the user needed to change the number in both the `version.rb` file, and a `Gemfile.lock` file. Commit `a086ca7` shows how the user corrected this error so that the repository is again passing the build (cf. annotation fix_1 in Fig. 2).

At the same time, another user submitted a pull request with the old version which split the `Gemfile.lock` into two separate files for different versions of the rails library. In the merge process, the user inadvertently removed the original `Gemfile.lock` without copying over the fix for the version number. The merge resulted in the repository taking commit `bce9420` and is in a similar state to the previous broken commit `10d78ad`. There are now two files that are not consistent with `version.rb`.

By analyzing the previous data from fix_1 (depicted in Fig. 2), VeriCI has learned that the `PATCH` value in the `version.rb`

is potentially problematic in this situation. Running VeriCI at the point of commit `bce9420` generates the error below:

```
Predicted build failure based on these keywords:
   - PATCH
   - simplecov
```

VeriCI predicted a build failure, and provides the explanation that this failure is likely due to one of the two listed error locations. We confirmed this error message as a true positive, as the next day the user discovered this fix on their own, and applied fix_2 to bring the repository back to a correct state [26].

III. Preliminaries

In this section we introduce the basic vocabulary in the continuous integration paradigm. We describe the formalism that we use to model the CI terms. This formalism is the basis for the future analysis and the learning process.

Repository Status. The core structures used in CI platforms are repositories. A repository, denoted by R, contains all information required for a CI build, such as source code, automated tests, lists of library imports, build scripts, and similar files. The CI process monitors how a repository evolves over time, so we use R_t to denote the state of repository R at time t. While many version control systems allow for a branching timeline (called a *branch* in git), we linearize and discretize that timeline according to the build order in the CI tool in accordance with the linearization of TravisTorrent [10]. Thus, the time indices are always non-negative, monotonically increasing integers. For a repository R_t, a typical CI tool usually has three possible outcomes:

1) All builds were successful and all tests pass
2) All builds were successful, but some tests fail
3) There was an error in the build process before tests could even be executed

Since the focus of this work is analyzing the CI *configuration* of the repository, we distinguish only between successful builds and failing builds. If all builds were successful (cases 1 and 2), there is no error in the CI configuration and we call the status of the repository "passing". We denote this build success with $S(R_t) = P$. In case 3, when a build failed, we call the status of the repository "failing" (or, a "failed build"), and we denote this with $S(R_t) = F$.

Of a particular interest are changes in the repositories that cause the build status to change, for example when the repository status changes from passing to failing. To capture this, we introduce the following notation:

$$S(R_{t,t+1}) = PF :\Leftrightarrow S(R_t) = P \wedge S(R_{t+1}) = F$$
$$S(R_{t,t+1}) = FP :\Leftrightarrow S(R_t) = F \wedge S(R_{t+1}) = P$$

Repository Summary. Our main algorithm for build prediction takes as input a dataset of feature vectors and labels. A feature vector is a set of tuples of the type $(\mathit{String}, \mathbb{R})$, and the label is the build status (passing or failing). To map a repository R_t to a feature vector, we introduce the notation of a repository summary, $\hat{R}_t$. A repository summary, $\hat{R}_t$, is feature vector capturing an abstraction of that repository's CI configuration.

We consider the scope of a CI configuration to encapsulate both the CI configuration files (e.g. `.travis.yaml`), as well as some parts of the source code configurations (e.g. library imports and their versions). There is no strict definition that separates the CI configuration and the application configuration or source code - they are intertwined by definition. The boundary of what we consider in the scope of CI configurations is defined by our feature extraction design. The process of extracting feature vectors that summarize the CI configuration is detailed in Sec. IV. We validate this definition empirically in Sec. V.

In the summarization process, we only select information that is relevant to the CI configuration. Every repository contains a number of files that are not relevant for deriving the properties about the CI configuration. Examples of these files include "readme" files, `.csv` files, or images. We filter out all such files based on their extensions. For example, for Ruby programs we consider all **.rb*, *Gemfile*, *gemspec* files.

IV. System Description

To statically check that a CI repository is correctly configured and the build will succeed, we must build a model of correctness. To do this, we analyze a number of existing CI repositories, including both the code and configuration files, and their corresponding CI build results. Once we learn this model of correctness, our system VeriCI can make a judgement on whether a new commit to the repository is likely to break the build. In the case that VeriCI predicts a build failure, VeriCI provides an explanation to the user for the prediction so that the user may proactively check and fix the CI configuration.

For each repository, we consider a training set of commits from the repository, R. The process of learning the model is an iterative process. As new commits appear in the repository, they are integrated into a refined version of the model.

The goal of producing useful error messages creates two new challenges in this domain. First, we must use a learning strategy that not only classifies the build status, but also provides a justification for output of the classifier. This problem of justification is referred to as *interpretability* [27] and *explainable AI* [28] in machine learning. In order to generate useful justifications, we must restrict ourselves from using any kind of metadata in our learning process. Our goal is to produce a justification that guides users to the root cause of a build failure. As such, the learning process must rely on code feature extraction that only uses a set of features directly tied to the code.

An overview of our learning approach is shown in Fig. 4 and consists of three main components:

1) We introduce *Abstraction Based Relabeling* (ABR) as a novel method to reduce noise in the training set $\mathcal{TR}$ of past commits. The underlying dataset is noisy due to non-deterministic build outcomes introduced by errors out of the users' control (such as network failures, power outages, or even a hardware failure on the part of the CI

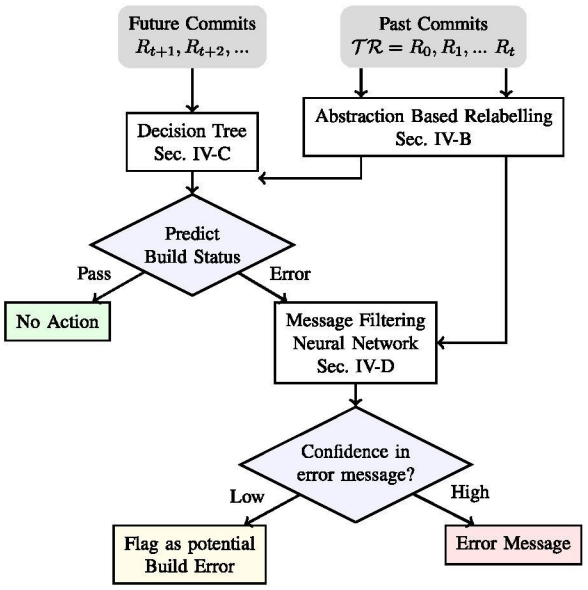

Fig. 4: VeriCI trains a model to predict build failures on previous commits. If the model predicts a new commit will result in an failure, VeriCI tries to generate an error message. If VeriCI has confidence in the error message, it reports this to the user, and otherwise remains silent.

provider). We call these types of errors *transient errors*. ABR ensures that if the extracted code feature vectors are the same, so are their build outcomes.

2) Given a pre-processed training set, we use decision trees to construct a model that predicts whether or not a build will fail. In the case that the decision tree predicts a build failure, we generate an error message.
3) Constructing an appropriate error message is difficult without actually running the build. To increase the confidence in providing a helpful message, we train a neural network model that allows us to filter out cases of messages that are likely not contributing in resolving the build failure.

A. Feature Extraction

The first step in our system design is to pre-process the training set of a timeline of past commits. This list of commits form our labelled dataset, with each past commit $R_0, \ldots, R_t \in \mathcal{TR}$ being tied to a label denoting the corresponding build status, as introduced in Sec. III. In order to leverage machine learning techniques, we must extract a selection of *code features* from these commits to form a summary of the repository state, $\hat{R}_t$ (also called a feature vector) to be used in the learning process.

We focus on extracting what we call *magic constant* code features. This feature tracks the use of hard-coded numerical constants in a repository. Magic constants are especially important in CI configurations as they often represent locations prone to misconfiguration, such as library version constraints. More generally, the core idea is that, if a line contains a scalar that we can embed into a feature vector, we build a tuple, where the first element is the context of the scalar, and the second element is the numerical constant value. We give an example of this process in Fig. 5.

Listing 5: R_1

```
import Tweet V1.0
m = rndMsg(1)
```

Listing 6: R_2

```
import Tweet V2.0
m = rndMsg(1)
```

Listing 7: $\hat{R}_1$

```
("import Tweet", 1.0)
("m = rndMsg", 1.0)
```

Listing 8: $\hat{R}_2$

```
("import Tweet", 2.0)
("m = rndMsg", 1.0)
```

Fig. 5: Two examples for extracting code features (Listings 7 and 8) from repository code (Listings 5 and 6 respectively) at time points 1 and 2 of a commit history.

The number of potential code features when analyzing a repository is intractably large (a complete feature extraction would encode all behavior of the code), thus we must reduce the dimensionality of the code by selecting key features. As with many feature extraction tasks, our code feature extraction uses a set of templates through a process known as feature engineering [29]. Our feature extraction templates are simply looking for numerical constants, and finding a corresponding keyword on the same line as that constant. To do this, we parse the relevant files in the codebase (R) to find lines of code that have numerical constants. Our simple parsing procedure handles common syntax patterns across languages such as ignoring comments, associating code across line breaks, parsing function call syntax, etc. (see link.anonymized for the exact code). There is no manual tuning of the features on a per repo, or per language basis. While this means we may consider magic constants that are specific to test or production code in our current work, we leave further filtering to future work.

B. Abstraction Based Relabeling

During this feature extraction process we may face cases where we end up with two repository summaries that are identical, although the build statuses differ. This can happen for two reasons. First, it is possible that our feature extraction has abstracted too much of the code and we have lost the key difference between a build that succeeds and a build that fails. Second, we must account for the fact that CI builds are not pure in the sense that some builds can fail due to transient errors. This non-determinism of configuration behavior has also been observed in other settings of configuration analysis [30]. Transient errors are problematic as having identical data points with different labels injects noise into the training set. In order to reduce the noise that our learning module needs to handle, we want to ensure that if extracted code feature vectors are the same, their build outcomes are also the same. That is, the goal of ABR is to ensure that $\hat{R}_t = \hat{R}_{t'} \Rightarrow S_{ABR}(R_t) = S_{ABR}(R_{t'})$, where $S_{ABR}(R_t)$ is the status of the repository at time t after relabeling.

We introduce the notion of *Abstraction Based Relabeling* (ABR) to account for the impact of CI build non-determinism

TABLE I: An example learning process demonstrating ABR. Source code for $\hat{R}_1$ and $\hat{R}_2$ are listed in Fig. 5.

Repository States	$\hat{R}_1$	$\hat{R}_2$	$\hat{R}_3$	$\hat{R}_4$
Original Status : $S(R_t)$	P	F	P	F
Code Features Extraction				
$\hat{R}$				
import Tweet	1.0	2.0	2.0	2.0
rndMsg	1.0	1.0	1.0	2.0
ABR Status : $S_{ABR}(R_t)$	P	P	P	F

and information loss during feature extraction. The goal of ABR is to relabel the training set so that it is self-consistent with our observation (feature extraction) of the training set. Initially, we start the ABR process with $S_{ABR}(R_t) = S(R_t)$ for all time points t. Then we examine each repository summary and if at some point the build status changes, but the code features do not, we relabel that failing status as a passing state. We only relabel failing statuses to passing (and passing to failing), as we want to minimize the noise in the positive classification set. Decreasing noise in the positive classification set (failing status) helps decrease false positives, although it does increase false negatives. In the context of CI prediction, we find this to be a positive trade-off in practice. Formally, the relabeling from the original build status $S(R_t)$ to the ABR build status $S_{ABR}(R_t)$, works as follows:

$$\text{If } S_{ABR}(R_{t,t+1}) = PF \wedge \hat{R}_t = \hat{R}_{t+1} \Rightarrow S_{ABR}(R_{t+1}) = P$$
$$\text{If } S_{ABR}(R_{t,t+1}) = FP \wedge \hat{R}_t = \hat{R}_{t+1} \Rightarrow S_{ABR}(R_t) = P$$

To understand the effect of ABR, we provide an example process in Table I. In this example, the build of $S(R_2) = F$ failed (for the sake of demonstration, due to a network failure), but we can see that $S(R_3) = P$ succeeded. During the learning process we do not have information on the root cause of these failures, but we can observe the code features and CI status, so we relabel $S(R_2)$ accordingly.

In addition to ABR, we tried applying resampling methods in VeriCI, such as SMOTE [31] to reduce the noise in the datasets. However, implementing oversampling by generating synthetic samples from the minority class adds noise to our training sets. While this assists in prediction, it decreases accuracy of error localization, which is the key contribution of this work. We also tried undersampling, but this led to deleting passing commits, which contain important information about the build fix. Hence, in order to balance both CI build status prediction accuracy and accurate error localization, we found that ABR was more effective than resampling methods.

C. Predicting Build Status

After we preprocess the training set with ABR, we build a decision tree classifier to analyze a new commit to a repository and predict the build status. Decision tree learning [32] is a widely used machine learning technique that constructs a tree consisting of two types of nodes: decision nodes and leaf nodes. Decision nodes contain a Boolean condition, which defines branching depending on if the condition is true or false. The leaf nodes represent the final classification of a data point. For a data point belonging to some leaf node in a decision tree, we extract a *decision path* that describes which decision nodes were used to arrive at that leaf node. This decision path is constructed by recording the traversal of the decision tree - that is, tracking the conditions in decision nodes (or their negations) on the path from the root to that leaf. More importantly, the ability of a decision tree to easily provide a justification for its classification is the key property of decision trees as a machine learning technique that makes it a good fit for our purposes.

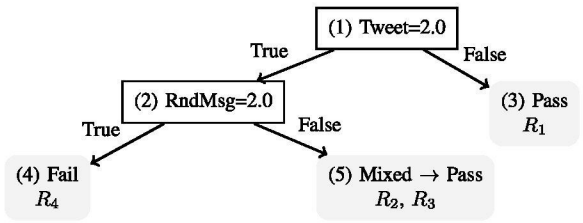

Fig. 6: An example of a decision tree with white boxes as decision nodes and gray boxes as leaf nodes.

Fig. 6 depicts a decision tree generated from the results of ABR shown in Table I. Our tree needs to classify four commits, based on the given two features (`Tweet` and `RndMsg`). In this example, the learning process results in a decision tree with three leaf nodes. Node (3) contains only $\hat{R}_1$, correctly classifying it as a passing node. Node (4) contains only $\hat{R}_4$ and it is correctly classified as a failing node. We call nodes (3) and (4) pure nodes, since they perfectly predict the commits from the training set.

In contrast, node (5) is a mixed node, since some repositories from the training set classified by this node pass ($S(R_2) = F$) and others fail ($S(R_3) = P$). Although these repositories had different build statuses, based on the feature vectors we have extracted, they are equivalent - and thus had equivalent ABR statuses. This leaf node cannot then be split further based on the available features and we cannot correctly classify both of these repositories. We mark both these builds as passing to avoid a false positive report where we tell the user there is an error, when in fact the system is passing.

Using decision trees as the classification learning technique also allows us to easily extract the root causes of the failed build. To do this, we trace the path traversed through the tree to classify a data point. As an example, we can construct an error message for the classification of R_4 by tracing the decision path and generating the error message: (`Tweet = 2.0` $\wedge$ `Rndmsg = 2.0`).

D. Filtering Error Messages

VeriCI predicts the build status of a new commit, and extracts a potential error message using the decision tree classification path. This is a key benefit to our technique, as these error messages allow VeriCI to not only pre-emptively tell users the status of their build, but also give them feedback

on how to fix it. However, not all our error messages are informative - it is possible that we may predict the correct status for the wrong reason. An incorrect error message (even if the build is failing) is potentially worse than no error message at all. If we report an incorrect error message, users may waste time searching in the wrong location for the root cause of the build error. However, if we can refrain from reporting bad error messages, even when the build does fail, the user workflow remains at the status quo as it is when not using VeriCI.

Thus, to ensure we minimize the number of misleading error messages, we trained a neural network to predict the effectiveness of our generated error messages. We measure the effectiveness of the error message by checking if the error message we report is contained within the `git diff` of the commit that eventually fixes the build. If any part of the decision tree path for the failing commit appears as a substring of the fix, we label the error message as Hit, otherwise we label the error message as a Miss.

In summary, we train our neural network with an input of labeled error message sets. To label our training set, we "learn from our mistakes" by looking at the error messages we have previously reported. In labeling our training set for the neural network, there are two cases we must consider. First, we may have previously reported an error, and have already seen the fixing commit for this failing build. In this case, we can label out error message by checking the ground truth. Second, we may have previously reported an error, but not yet seen a fixing commit for the failing build. In this case, we will construct a label instead from the breaking commit, e.g., the `git diff` between this failing build and the last previous build that succeeded.

In order to encode the error message as a feature vector that can be understood by the neural network, we convert the text to its ASCII representation. As error messages can differ in length, we pad all error messages shorter than a fixed feature vector size with spaces, and truncate all error messages longer than this size. We then use a technique called *feature scaling* to ensure all features (characters of the error message) are represented by a float value 0-1. This is an important step, as neural networks are sensitive to the scale of feature values, whereas decision trees are not. More specifically, we used a multi-layer perceptron neural network with two hidden layers. Finally, to integrate the neural network with the rest of VeriCI, we use the resulting model as a binary classifier. If the neural network predicts an error message to be a Hit, we report both the predicted build status and its corresponding error messages to the user. However, if the neural network predicts a Miss, we only report the predicted build status to the user. We report the frequency with which we accurately predict Hits and Misses in Sec. V-B.

V. Evaluation

We implemented VeriCI to statically check repositories that use the open-source continuous integration testing tool, TravisCI. TravisCI is an ideal testbed for analyzing CI builds, as the tool is free for open-source projects and widely used on GitHub. Additionally, a log history of the builds for a number of large, active, and open-source project has been made available through the TravisTorrent dataset [10]. This dataset directly provides us with the labeled, temporally ordered commit data over many repositories. Our evaluation set consists of all 35 repositories available in TravisTorrent which have 150 - 200 commits and whose main language is Ruby. We chose Ruby as this is the language with the best representation in the TravisTorrent dataset. As we require many commits as training data, we wanted to take repositories with the largest number of commits possible. However, taking repositories with a history of larger than 200 commits slowed down the training process, due to limited network speed and memory needed to clone the repositories.

In the evaluation, we aim to answer the following questions:

1) Does VeriCI accurately predict the build status of a commit before the build actually executes?
2) Does VeriCI report error messages that correctly identify the root cause of the build failure?
3) Are the error messages VeriCI generates helpful to users?

A. Accuracy of Prediction

In order to evaluate the accuracy of VeriCI, our evaluation models the way VeriCI would be used in a real world setting. In practice, when integrated into the DevOps pipeline, VeriCI would be trained over a set of commits up until the present moment, and used to predict commits for one day. Then, at the end of the day, VeriCI would build a new model, incorporating the information from that day, so that the model the next day can learn from the new commits and any mistakes that were made. We show this on a timeline in Fig. 7.

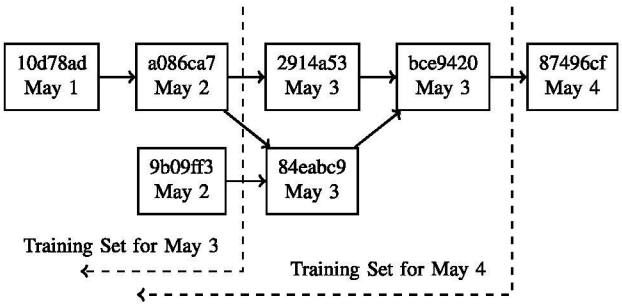

Fig. 7: VeriCI rebuilds a model at the end of every day to provide better prediction the next day.

The strategy that we employed in the evaluation is presented in Table II. For a repository with n total commits, we build the first model with $n/2$ commits, and use that model to evaluate commit $R_{(n/2)+1}$. We then rebuilt the model with a training set of $(n/2)+1$ commits and used that model to predict the status of the next commit. Using this approach, we found that VeriCI can predict build status with an overall accuracy of 91%. This is competitive with other tools for build status prediction based on metadata such as the committer's historical rate of correct

commits, size of the commit, sentiment analysis on the commit message, use of emojis, and similar metadata [14], [12], [15], [13], [18], [17]. We also find VeriCI has a F1-score of .569, though existing tools do not report this metric, making it difficult to compare. We also attempted to run the existing tools for build prediction (HireBuild [17] and HoBuFF [18]), but the tools were not made publicly available in a way that allowed use to rerun their tool (HireBuild is distributed as a .jar without source code that only runs their original experiments, while the link for the tool in HoBuFF paper is a 404).

One interesting note is that we should always expect our prediction to have a non-zero number of false negatives and false positives. A recent study investigated the reliability of TravisCI builds [30], and found that roughly 9% of builds have a misleading or incorrect outcome. These incorrect outcomes manifest from transient errors outside the code base, such as in dependencies or through network failures. Since our method only looks at the code inside the repository itself, and thus learns from this "misleading" build data, we should not expect that VeriCI will be able to detect these types of errors.

B. Accuracy of Error Messages

We have shown that VeriCI can predict build status with very high accuracy, but the key innovation in this work is that we also provide an explanation for the classification that our model provides. We must also evaluate how well the explanation provided by VeriCI corresponds to a change in the code base that would successfully fix the failing build status. To evaluate the rate at which VeriCI provides accurate error messages, we check if any of the keywords we presented in the error message appear in the difference between the failing commit and the next passing commit. If the user has changed a keyword that we suggested in order to fix the repository's build status, it can be seen as evidence that we suggested a correct root cause of the failure.

Using this metric, we found that over the repositories listed in Table II, the error we reported corresponded to the change that the user eventually made to fix the build 75% (15/20) of the time, for the rules we learned. We report the full effect of the error messaging filtering technique described in Sec. IV-D in Table III. We recorded the results of running VeriCI with various configurations of ABR to measure the impact of these new methods. Enabling ABR has little effect the accuracy of our error messages (increasing from %90.55 to %90.96), and reduces the average error message size from 2.31 to 1.99. This means that average number of keywords reported in an error message (the decision path depth) by VeriCI was 1.99. In other words, VeriCI correctly identifies a critical key word in a breaking commit with an average of 1.99 guesses in 75% of cases.

C. User Study

In the previous section we have shown that the error messages produced by VeriCI are accurate, but this still leaves the question of how useful these messages are to developers. To assess the impact of VeriCI of how developers identify and fix CI build failures, we conducted a user study with a total of 20 professional developers. Our goal was to investigate whether the use of VeriCI increases the correctness of solutions to errors in CI, compared to solely the output of TravisCI. The null hypothesis and alternative hypothesis corresponding to this question can be formulated as follows:

- $H1_0$: Error messages generated by VeriCI *do not* increase the user's ability to correctly find solutions to CI errors.
- H1: Error messages generated by VeriCI increase the user's ability to correctly find solutions to CI errors.

Tasks & Environment. We asked participants to fix two TravisCI build failures from a GitHub repository [33], [34]. Given the failed builds, participants were asked to identify the root cause of the failure and describe a possible fix. Our user study was designed as a between-subjects study. In the control group, participants were required to fix builds with the log file provided by TravisCI itself. In the treatment group, we provided the error messages generated from VeriCI. In both groups, participants had no prior knowledge of the code base and were told to make use of any resources available to them (including internet), except looking at future commits in the repository of the study. This setting was designed to, as closely as possible, model the information available to developers in real-world CI use cases. Users were asked to identify the root cause and propose a fix in an open response field.

Variables & Analysis. The independent variable in our experiment is the presence of error messages by VeriCI during the tasks. The dependent variable is the correctness of the given answers. To convert the participants' answers into quantitative information for further analysis, we designed a rubric to classify answers into one of three categories (following the methodology by similar user studies [35], [36]). We evaluate participants' results by comparing their answers to the actual subjects' fix provided by the TravisCI users.

1) The participant identifies the root cause or describes a fix with enough detail to directly resolve the build failure.
2) The participant mentions part of the root cause or describes a partial fix, but does not provide enough detail to completely resolve the failure. We use a similar, but more relaxed, criteria for correctness here compared to Sec. V-B - the user must only mention a substring of any possible fix (not just the eventual fix in this repository).
3) The participant incorrectly identifies the root cause and proposes an incorrect fix.

Subjects. Since our study setting requires knowledge on how to identify and fix production build errors, we recruited 20 subjects ($N = 20$) with professional software development experience to participate in our study, 10 for both control and treatment conditions. We also collected the subjects' levels of expertise on different topics (continuous integration, TravisCI, Ruby, and GitHub) through a self-reported assessment on a Likert-scale. We performed a Mann-Whitney U test [37] to assess whether both groups' expertise is comparable. The results did not show any significant differences between expertise levels in the groups (results were overall not significant at

TABLE II: Classification rates across 150 - 200 commits of a random selection of repositories including TP (True Positive - correctly predicted failure), TN (True Negative), FP (False Positive - incorrect predicted failure when actual status was pass), and FN (False Negative). We also report precision (= TP / TP + FP) and recall (= TP / TP + FN).

Repository Name	Accuracy	Precision (TP / TP + FP)	Recall (TP / TP + FN)	TP	TN	FP	FN
sferik/rails_admin	93%	0.933	0.700	14	75	1	6
thoughtbot/shoulda-matchers	98%	1.000	0.500	2	80	0	2
jnunemaker/flipper	87%	0.850	0.630	17	68	3	10
pagseguro/ruby	91%	0.750	0.462	6	80	2	7
activescaffold/active_scaffold	89%	0.778	0.583	7	51	2	5
twitter/twitter-cldr-rb	92%	0.833	0.870	20	63	4	3
Average over 35 repositories	91.0%	0.780	0.448	2.943	41.914	0.829	3.629

TABLE III: Accuracy of neural network postprocessing on error messages. N/A indicates we do not report a message to the user.

Prediction	Ground Truth	Number of Occurrences
Hit	Hit	15
Hit	Miss	5
Miss	Hit	5
Miss	Miss	37
Prediction	**Reason for N/A**	**Number of Occurrences**
N/A	Non-Alphanumeric	1
N/A	Insufficient Training Data	24
N/A	No Fixing Commit	16

TABLE IV: Statistical results related to the correctness of how developers identify and fix CI build failures

Task One	Exact Fix	Partial Fix	Not Related
Control Group ($n_C = 10$)	1	0	9
Treatment Group ($n_T = 10$)	2	4	4
Task Two	**Exact Fix**	**Partial Fix**	**Not Related**
Control Group ($n_C = 10$)	6	2	2
Treatment Group ($n_T = 10$)	4	4	2

$p < .05$ with critical value of $U > 27$, full table available in our online appendix).

Results. We asked each participant to fix two failed TravisCI builds. We report the detailed distribution of their coded answers in Table IV. We combined the category one and two into a new category of "at least partially correct" and use Barnard's test[2][38] for significance. We found strong evidence ($p = 0.0227$) that rejects the null hypothesis (i.e. we showed that VeriCI helped participants to correctly identify the root cause or find a fix for the build failure). This experiment strongly indicates that VeriCI successfully helped users to find the root cause of the failed build or even a fix. For the second task, we found no significant differences in both control and treatment conditions. We found that 80% participants (eight out of ten) achieved at least a partially correct fix.

We examined the second task more carefully to form a hypothesis for this inconclusive result. When checking the Travis log file of this specific build, we found that the root cause of this build failure was already contained in the Travis log file. This observation leads us to the theory that if the root cause is easily extractable from the log file, the generated error message by VeriCI does not yield any significant differences to find the fix.

[2]A more conservative version of Chi-Square specialized to small values.

TABLE V: Training time for VeriCI over a single repository with variable number of commits (training set size).

Num of Commits	1	20	40	60	80
Training Times (s)	0.049	1.067	2.887	4.824	7.623

VI. Discussion

We discuss the challenges of building learning-based static analyzers, and how we mitigate possible threats to the validity of our evaluation.

Uncertainty and Explainability. We envision VeriCI being used by a developer, or team of developers, in their everyday CI workflow. In deploying probabilistic program analysis tools, one of the most important metrics is that the tool has a very low false positive rate. In the context of VeriCI, it quantifies the number of times developers are falsely led to believe there is a build failure that they need to investigate. Often in practice, this can be the single metric that prevents adoption, as users will quickly ignore tools with too many false alarms. Our evaluation results (Table II) have shown that the average number of false positive reports to average number of correct classifications are at only 1.8% (FP/TP+TN). For the remaining uncertainty, our use of decision trees for learning provides developers with a comprehensible set of rules (see Fig. 6) that led to the classification. This allows developers to reason about the outcome more quickly and disqualify false positives and continue to run the build on the server.

Scalability. We find that VeriCI scales roughly linearly (dependent on the size and complexity of the repositories in the training set) and list training times in Table V. All the experiments in this section are conducted on a MacBook Pro equipped with Haswell Quad Core i7-4870HQ 2.5 GHz CPU, 16GB memory, and PCIe-based 512 GB SSD hard drive.

A. Threats to Validity

Internal Validity. Our learned model may not be the most ideal within the space of potential hypotheses that classify

build outcomes. In particular, there may be more effective ways to do code feature extraction. However, the goal of VeriCI is not to act as a standalone tool, but rather interface with existing build error prediction tools. The focus of VeriCI is on prediction of build error *localization*, not just prediction of build error *status*. Despite this more challenging setting, overall, our results show we have found a sufficiently good model to demonstrate the value of using a restricted set of features in the learning process.

External Validity. Our evaluation samples from open-source projects using Ruby with TravisCI. We sought to ensure that our results generalize to other contexts, such as repositories that focus on different core programming languages. First, we note that the features that we use for learning are agnostic to the specific languages and technologies used. We intended for our *magic constant* feature extraction to be easily applied to different programming and configuration languages. To test this, we ran VeriCI on 22 Java repositories (selected using the same criteria described in Sec. V) and found VeriCI to have 90.6% predication accuracy, but leave a more thorough investigation to future work. In terms of generalizing to other CI frameworks beyond Travis, we note that the information required from TravisCI to build our model is also available in other continuous integration frameworks.

VII. Related Work

Building tools for configuration support, management, and verification has been an active area of research [39], [40], [41], [42], [43], [44], [21], [45], [46], [22]. Nevertheless, generating models and checking configuration settings still remains an open problem.

Continuous Integration build prediction. The increasing prevalence of CI as a core software development tool has inspired significant work on the topic. The topic of CI analysis in particular was explored in the 2017 Mining Software Repositories Mining Challenge [47]. Existing efforts on prediction of build failures for CI largely rely on metadata in the learning process [12], [13], [14], [17], [18]. Natural language processing and sentiment analysis has also been used to predict build status [15]. Other work falls into the category of test-case prioritization, and is used to skip CI builds in part or in whole by analyzing prior build results [48], or observing changes to comments [49]. However, as none of these metrics are tied to the code itself, approaches using such metadata cannot provide the user with any information about which part of the code they should inspect to isolate and fix the root cause of the failure. That is - the prior work only provides *file level* fault localization [16], [17], not *code level* fault localization. In contrast, VeriCI specifically predicts the build status based on direct code features that users have control over, allowing them to more easily fix builds.

An additional line of work in CI, complementing build failure prediction and localization, is that of post-mortem analysis to automatically repair failed builds. As one example, BuildMedic [50], is able to automatically repair broken CI builds in Java repos that use the Maven build system by analyzing the build logs, but requires the build to execute in order to analyze the logs.

Learning-based configuration verification. Several machine learning based misconfiguration detection efforts also have been proposed [51], [20], [52], [53], [54], [55]. EnCore's [55] learning process is guided by a set of predefined rule templates that enforce learning to focus on patterns of interest. These works largely target key-value style configurations, which is a more structured representation than CI configurations.

Language-support misconfiguration checking. There have been several language-based efforts proposed for specifying the correctness of system-wide configurations. For example, in the datacenter network management field, the network administrators often produce configuration errors in their routing configuration files. PRESTO [56] automates the generation of device-native configurations with configlets in a template language. Loo *et al.* [57] adopt Datalog to reason about routing protocols in a declarative fashion. COOLAID [58] constructs a language abstraction to describe domain knowledge about network devices and services for convenient network management. In software configuration checking area, Huang *et al.* [59] proposed a specification language, ConfValley, for the administrators to write their specifications, thus validating whether the given configuration files meet administrators' written specifications. ConfSuggester [22] allows software versions to be updated while making appropriate changes to configuration options to maintain backwards compatibility.

Misconfiguration diagnosis. Misconfiguration diagnosis approaches have been proposed to address configuration problems post-mortem. For example, BART [60] automatically extracts anti-patterns and decay in CI by analyzing regular build logs and repository information, and supports build fixing through CI build log summarization and linking to Stack Overflow resources. In addition to log analysis, ConfAid [40] and X-ray [61] use dynamic information flow tracking to find possible configuration errors that may have resulted in failures or performance problems.

VIII. Conclusions

This paper has presented a novel tool, VeriCI, that automatically checks for errors in CI configurations before the build process. Driven by the insight that repositories in CI environments are already labeled with build status histories, our approach automatically generates specifications for correct CI configurations. We evaluate VeriCI on real world data from GitHub and find that we have 91% accuracy of predicting a build failure. We believe this tool presents a step forward in improving programmer productivity by eliminating the time spent waiting on unsuccessful builds.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No.s CCF-1715387, CCF-2105208, CCF-1553168 and CNS-1565208.

REFERENCES

[1] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *ICSE*. ACM, 2014, pp. 345–355.

[2] "Travis CI," https://travis-ci.org/, Jan. 2021.

[3] "CircleCI," https://circleci.com/, Jan. 2021.

[4] "Jenkins," https://jenkins.io/, Jan. 2021.

[5] "GitLab CI," https://about.gitlab.com/, Jan. 2021.

[6] "Codefresh," https://g.codefresh.io/signup?ref=BJV2J4zib, Jan. 2021.

[7] "TeamCity," https://www.jetbrains.com/teamcity/, Jan. 2021.

[8] SuperCollider, "Supercollider builds," Oct. 2021, https://travis-ci.org/github/supercollider/supercollider/builds.

[9] N. Bjorner, "Z3 azure build," Oct. 2021, https://dev.azure.com/Z3Public/Z3/_build/results?buildId=2507&view=results.

[10] Z. A. Beller M, Gousios G, "Oops, my tests broke the build: An analysis of travis ci builds with github," PREPRINT, 2016. [Online]. Available: https://doi.org/10.7287/peerj.preprints.1984v1

[11] SuperCollider, "[sc-dev] 3.9 delayed to tomorrow," Mailing List, Jan. 2018, http://www.listarc.cal.bham.ac.uk/lists/sc-dev-2018/msg57997.html.

[12] A. Ni and M. Li, "Cost-effective build outcome prediction using cascaded classifiers," in *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 2017.

[13] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting build failures using social network analysis on developer communication," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[14] A. E. Hassan and K. Zhang, "Using decision trees to predict the certification result of a build," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006.

[15] K. V. R. Paixão, C. Z. Felício, F. M. Delfim, and M. de A. Maia, "On the interplay between non-functional requirements and builds on continuous integration," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017. [Online]. Available: https://doi.org/10.1109/MSR.2017.33

[16] F. Hassan, "Tackling build failures in continuous integration," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019.

[17] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *International Conference on Software Engineering (ICSE)*, 2018.

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

[19] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic automated language learning for configuration files," in *CAV*, Jul. 2016.

[20] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac, "Synthesizing configuration file specifications with association rule learning," *PACMPL*, vol. 1, no. OOPSLA, 2017. [Online]. Available: http://doi.acm.org/10.1145/3133888

[21] A. Weiss, A. Guha, and Y. Brun, "Tortoise: interactive system configuration repair," in *ASE*, 2017. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115673

[22] S. Zhang and M. D. Ernst, "Which configuration option should I change?" ser. ICSE, 2014.

[23] d. bbenezech, "Railsadmin," https://github.com/sferik/rails_admin, 2016, 3fd3b32551e6d2d41cbbc623c99d08656a80be07.

[24] mshibuya, "Rails admin," https://github.com/sferik/rails_admin, 2016, dca8911f240ea11ebb186c33573188aa9e1b031d.

[25] scambra, "Active scaffold," https://github.com/activescaffold/active_scaffold, 2016, 10d78ad6ac45a0a55e3c15e12c39d2019aff5146.

[26] ——, "Active scaffold," https://github.com/activescaffold/active_scaffold, 2016, 87496cfa09a49d071817ba3da0fa6364c92a5191.

[27] G. Forman, "An extensive empirical study of feature selection metrics for text classification," *Journal of machine learning research*, vol. 3, no. Mar, 2003.

[28] D. Gunning, "Explainable artificial intelligence (XAI)," *Defense Advanced Research Projects Agency (DARPA), nd Web*, 2017.

[29] S. Scott and S. Matwin, "Feature engineering for text classification," in *16th International Conference on Machine Learning (ICML)*, Jun. 1999.

[30] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh, "Noise and heterogeneity in historical build data: An empirical study of travis ci," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238171

[31] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *J. Artif. Int. Res.*, vol. 16, no. 1, Jun. 2002.

[32] L. Rokach and O. Maimon, *Data Mining with Decision Trees: Theroy and Applications*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2008.

[33] jnunemaker, "Flipper," https://github.com/jnunemaker/flipper, 2019, f4d68e4eb923d1bd6273aa452fbc4dd7146db75f.

[34] ——, "Flipper," https://github.com/jnunemaker/flipper, 2019, 9221c50d83e7214fad8f971e5d8d6b76453ebd4f.

[35] B. Cornelissen, A. Zaidman, A. Van Deursen, and B. Van Rompaey, "Trace visualization for program comprehension: A controlled experiment," in *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 2009.

[36] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: A controlled experiment," in *ICSE*. IEEE, 2011.

[37] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[38] G. A. Barnard, "A new test for $2\times$ 2 tables," *Nature*, vol. 156, 1945.

[39] T. Dai, A. Karve, G. Koper, and S. Zeng, "Automatically detecting risky scripts in infrastructure code," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, 2020, p. 358–371.

[40] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *OSDI*, Oct. 2010.

[41] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci, "Usable declarative configuration specification and validation for applications, systems, and cloud," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*. ACM, 2017.

[42] M. Raab, "Elektra: universal framework to access configuration parameters," *The Journal of Open Source Software*, vol. 1, no. 8, 2016.

[43] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: a configuration verification tool for puppet," in *PLDI*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908083

[44] Y. Su, M. Attariyan, and J. Flinn, "AutoBash: Improving configuration management with operating systems," in *21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[45] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *OSDI*, Nov. 2016.

[46] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Comput. Surv.*, vol. 47, no. 4, p. 70, 2015.

[47] J. M. González-Barahona, A. Hindle, and L. Tan, Eds., *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 2017. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=7959735

[48] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *FSE*, 2014. [Online]. Available: https://doi.org/10.1145/2635868.2635910

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

[50] C. Macho, S. McIntosh, and M. Pinzger, "Automatically repairing dependency-related build breakage," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

[51] B. Amand, M. Cordy, P. Heymans, M. Acher, P. Temple, and J.-M. Jézéquel, "Towards learning-aided configuration in 3d printing: Feasibility study and application to defect prediction," in *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 2019.

[52] M. Santolucito, E. Zhai, and R. Piskac, "Probabilistic automated language learning for configuration files," in *28th Computer Aided Verification (CAV)*, Jul. 2016.

[53] O. Tuncer, N. Bila, C. Isci, and A. K. Coskun, "Confex: An analytics framework for text-based software configurations in the cloud," Tech. Rep. RC25675 (WAT1803-107), IBM Research, Tech. Rep., 2018.

[54] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y. Wang, "Automatic misconfiguration troubleshooting with PeerPressure," in *OSDI*, Dec. 2004.
[55] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, "Encore: Exploiting system environment and correlation information for misconfiguration detection," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2014.
[56] W. Enck, P. D. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. G. Greenberg, S. G. Rao, and W. Aiello, "Configuration management at massive scale: System design and experience," in *USENIX Annual Technical Conference (USENIX ATC)*, Jun. 2007.
[57] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan, "Declarative routing: Extensible routing with declarative queries," in *ACM SIGCOMM (SIGCOMM)*, Aug. 2005.
[58] X. Chen, Y. Mao, Z. M. Mao, and J. E. van der Merwe, "Declarative configuration management for complex and dynamic networks," in *ACM CoNEXT (CoNEXT)*, Nov. 2010.
[59] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou, "Confvalley: A systematic configuration validation framework for cloud services," in *EuroSys*, Apr. 2015.
[60] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall, "Every build you break: Developer-oriented assistance for build failure resolution," 2020.
[61] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *OSDI*, Oct. 2012.