RavenBuild: Context, Relevance, and Dependency Aware Build Outcome Prediction

GENGYI SUN, University of Waterloo & Ubisoft La Forge, Canada
SARRA HABCHI, Ubisoft La Forge, Canada
SHANE MCINTOSH, University of Waterloo, Canada

Continuous Integration (CI) is a common practice adopted by modern software organizations. It plays an especially important role for large corporations like Ubisoft, where thousands of build jobs are submitted daily. Indeed, the cadence of development progress is constrained by the pace at which CI services process build jobs. To provide faster CI feedback, recent work explores how build outcomes can be anticipated. Although early results show plenty of promise, the distinct characteristics of Project X—a AAA video game project at Ubisoft—present new challenges for build outcome prediction. In the Project X setting, changes that do not modify source code also incur build failures. We also observe that the code changes that have an impact that crosses the source-data boundary are more prone to build failures than code changes that do not impact data files. Since such changes are not fully characterized by the existing set of features for build outcome prediction, state-of-the-art models tend to underperform.

To incorporate the data context, we propose RavenBuild—a novel approach to build outcome prediction that leverages context-, relevance-, and dependency-aware features. In the Project X context, we observe that RavenBuild improves the F1-score of the failing class by 50%, the recall of the failing class by 105%, and the AUC by 11% with respect to the state-of-the-art BuildFast approach. To ease adoption in settings with heterogeneous project sets, we also provide a simplified alternative RavenBuild-CR, which excludes dependency-aware features. We observe across-the-board improvements when RavenBuild-CR is applied to 22 open-source projects and Project X. On the other hand, we find that a naïve Parrot approach, which simply echoes the previous build outcome as its prediction, is surprisingly competitive with BuildFast and RavenBuild. Though Parrot fails to predict when the build outcome differs from their immediate predecessor, Parrot serves well as a tendency indicator of the sequences in build outcome datasets. Thus, we recommend that future studies also compare to the Parrot approach as a baseline when evaluating build outcome prediction models.

CCS Concepts: • **Software and its engineering → Empirical software validation**; **Software testing and debugging**; **Maintaining software**.

Additional Key Words and Phrases: continuous integration, build outcome prediction, maintenance cost, mining software repositories

ACM Reference Format:
Gengyi Sun, Sarra Habchi, and Shane McIntosh. 2024. RavenBuild: Context, Relevance, and Dependency Aware Build Outcome Prediction. *Proc. ACM Softw. Eng.* 1, FSE, Article 45 (July 2024), 23 pages. https://doi.org/10.1145/3643771

Authors' addresses: Gengyi Sun, University of Waterloo & Ubisoft La Forge, Waterloo & Montréal, Canada, gengyi.sun@uwaterloo.ca; Sarra Habchi, Ubisoft La Forge, Montréal, Canada, sarra.habchi@ubisoft.com; Shane McIntosh, University of Waterloo, Waterloo, Canada, shane.mcintosh@uwaterloo.ca.

ACM 2994-970X/2024/7-ART45
https://doi.org/10.1145/3643771

1 INTRODUCTION

Continuous Integration (CI) [6] enables rapid feedback by invoking build and test jobs for the change sets that development teams produce. For large organizations, adopting CI can accelerate development [38], but it is not without costs. The execution of a slow CI job can delay time-to-feedback [11], forcing developers to perform a mentally taxing context switch [41] to another task or remain effectively idle. Moreover, CI execution consumes a large quantity of computational resources [16, 17], which can quickly accrue annual costs on the order of millions of dollars [18].

Ubisoft heavily invests in the execution of builds for its games. For example, for Project X (the latest installment in a ten-year video game franchise), thousands of build jobs are submitted daily, consuming an average of 49 build hours on the main branch. The build duration for Project X ranges between 10 to 30 minutes, delaying developers from receiving immediate feedback and incurring expensive computational costs. Solutions that leverage build outcome prediction [14] has been proposed to address these limitations. Those solutions may reduce CI feedback time by reporting predicted outcomes to developers before the build has been performed [1, 3, 15, 22, 32, 35, 36]. Solutions may also save computational resources by skipping CI builds that are deemed unnecessary [1, 2, 21–24]. Since both contributors to open-source communities [3] and developers at Ubisoft have expressed concerns about the stability of CI after adopting a skipping strategy, our goal in this paper is to reduce CI feedback time.

Although early results show plenty of promise, the distinct characteristics of Project X present new challenges for build outcome prediction. Prior work on build outcome prediction has largely focused on open-source projects that are code-intensive. As such, many features adopted by these studies are code-specific, *e.g.,* the number of lines changed in source code. In the Project X setting, data artifacts and changes therein are more prevalent than source code and code changes. These data artifacts play a crucial role in the user experience when playing the game. The game engine compiles the data artifacts with source code in an order-sensitive manner that respects the specified dependencies. Therefore, if data artifacts are corrupted, or dependencies are not respected, data changes will incur build failures.

To assess the code and data dependencies, we construct a multidisciplinary dependency graph [37] that connects code, data, and computational nodes that enable interactions between them (*i.e.,* boundary nodes). Our analysis of the multidisciplinary dependency graph of Project X shows that code changes that have an impact that crosses the code-data boundary are more prone to build failures than code changes that do not impact data files. These risky cross-boundary changes also impact a large number of nodes that depend on them within the graph. The method by which we analyze the multidisciplinary dependency graph forges a new direction for extracting features for build outcome prediction. We also believe that this sort of change impact analysis will generalize to other (non-game) settings.

To incorporate dependency data, we propose RavenBuild—a novel approach to build outcome prediction that considers three families of file type-agnostic features when making predictions: (1) *context-aware* features that characterize the intention and location of the change being submitted; (2) *relevance-aware* features compare the current build and its immediate predecessor to provide indicators that suggest when the model should consider or ignore metadata about the previous build; and (3) *dependency-aware* features that assess the impact of the change on other files in the dependency graph. We apply the state-of-the-art BuildFast model [3] and our proposed RavenBuild enhancements to Project X, and observe that RavenBuild improves BuildFast by 50% in the F1-score of the failing class, 105% in the recall of the failing class, and 11% in AUC.

While our results indicate that dependency-aware features are valuable for build outcome prediction and can generalize to non-game settings, extracting them requires extensive project-specific knowledge. Therefore, to ease adoption in settings with heterogeneous project sets, we also provide a simplified alternative, RavenBuild-CR, which only relies on context- and relevance-aware features. We apply RavenBuild-CR to 22 open-source projects and Project X, and observe across-the-board improvements as well.

Surprisingly, we find that although both BuildFast and RavenBuild have promising performance, they cannot consistently outperform a naïve Parrot approach, which simply echoes the previous build outcome as its predictions. In a project-level comparison, even though Parrot does not outperform BuildFast and RavenBuild across all of the evaluation metrics, Parrot outperforms BuildFast and RavenBuild in, respectively, 6 and 4 of the 23 studied projects with respect to 3 of the 4 evaluation metrics. While Parrot outperforms the state of the art, it fails to capture the cases where build outcomes flip from passing to failing or failing to passing. These are the cases that build outcome prediction models provide the most value for developers. Nevertheless, the surprising performance of Parrot illustrates natural tendencies in build outcome prediction datasets. Since build outcomes tend to occur in sequences, we recommend that future studies compare with the Parrot approach as a baseline when evaluating build outcome prediction models.

2 RELATED WORK

In this section, we situate our work with respect to the literature on the benefits and costs of CI (Section 2.1), the sets of features that are typically used for build outcome prediction (Section 2.2), and work that leverages dependency analytics to improve developer experience (Section 2.3).

2.1 Benefit and Cost of CI

CI is widely adopted in modern software organizations [19] to automate integration and release routines [26, 38]. Prior work reports that developers face trade-offs [17] when adopting CI. As the software evolves, the cost of maintaining [28] and executing [19] build tools tend to grow. Therefore, researchers have invested in reducing the time-to-feedback and resource consumption.

To accelerate CI feedback, build outcome prediction has been proposed [15, 35, 36]. Hassan and Zhang [14] use decision trees to predict the outcome of certification testing. To conserve computational resources, Jin and Servant [2, 21, 22, 24] propose to skip passing builds using machine learning classifiers. A less aggressive approach is to select and only (re-)execute the tests that failed in prior builds [7, 16, 29, 34] and/or skip the tests that are unaffected (i.e., likely to pass) [8]. Gallaba *et al.* [9] propose Kotinos—a CI service that skips unaffected build steps in a technology-agnostic fashion. Jin and Servant also propose HybridCISave [23], which skips entire builds, as well as tests within builds, that are anticipated to pass.

The context-, relevance-, and dependency-aware features that we propose in this paper complement and enhance build outcome prediction. In addition, while prior work has focused on the code and test aspects of change sets, to the best of our knowledge, this paper is the first to expand build outcome prediction to accommodate changes to data files.

2.2 Build Prediction Features

Features that are extracted for build outcome prediction can be classified into those that characterize: (1) the current build, (2) the previous build, (3) historical trends, and (4) the relevance between the previous and current build. Hassan and Wang [15] use features that characterize current and previous builds. Chen *et al.* [3] use current, previous, and historical features to provide faster time-to-feedback. To execute as many failing builds as early as possible, Jin and Servant [22] propose SmartBuildSkip, which predicts the first builds in a sequence of build failures and the consequent

build failures separately with current build data. To the best of our knowledge, only Ni and Li [32] use the relevance features and refer to them as "connection to last push". Our proposed RavenBuild features differ in that RavenBuild characterizes the change-level relevance between the current build and its immediate predecessor, rather than the status of source or configuration files in the previous build. While the two sets of features have names that sound similar, the relevance features of RavenBuild are fundamentally different from Ni and Li's "connection to last push".

In addition to features for build outcome prediction, Abdalkareem *et al.* [1, 2] use CI-skip rules to determine which builds can be entirely skipped. PreciseBuildSkip [24] also proposes CI-run rules that supersede skip rules for safety improvements. PreciseBuildSkip is not a suitable baseline for video-game repositories like Project X. Indeed, 61% of the build failures of Project X are due to changes made solely to data files, which are not well-handled by the ruleset of PreciseBuildSkip. For example, according to the rule "NoSrcFileChange", all data changes should be skipped. Developing such rules for the data files would be impractical because there are numerous types of data files, each requiring deep subject matter expertise to formulate effective and safe CI-skip and CI-run rules.

2.3 Dependency Graph Analysis

Dependency graph information is valuable for research on CI because it contains rich data about how artifacts are depending on each other, which can be used to reason about build behaviour. In our prior work [37], we analyzed video game dependency graphs, demonstrated the prevalence of changes that have cross-boundary impact, as well as their tendency to incur build breakages and induce future fixes. In terms of CI acceleration, Kotinos [9] infers file dependencies and identifies unaffected files and artifacts, avoiding unnecessary build activity for subsequent builds. Zimmermann and Nagappan [40] use graph complexity metrics that were extracted from dependency graphs to predict subsystem failures.

Dependency analysis has also been leveraged for test case selection. Gligoric *et al.* [12] developed Ekstazi, which analyzes dependencies to select influenced tests for execution. The file-level dependencies captured in the dependency graph allow for effective regression test selection. Gligoric *et al.* also conducted experiments to compare dependency granularities at the method, class, and file levels, concluding that using the file-level dependencies (*i.e.,* the same granularity that we use in our multidisciplinary dependency graph) is the safest because it also captures external file dependencies. In contrast, Gligoric *et al.* discussed the limitations of FaultTracer [39]—a test case selection approach that uses an extended call graph to identify suspicious changes and the tests that they influence.

3 CHARACTERISTICS AND CHALLENGES OF THE VIDEO GAME SETTING

In this section, we describe the Project X setting (Section 3.1) and the challenges that it presents for build outcome prediction (Section 3.2).

3.1 Characteristics of Project X

Project X is the most recent installment of a popular video game in a franchise that was first released more than ten years ago by Ubisoft. Project X contains millions of production files, including source code, graphical assets, and audio samples. Throughout its development history, more than 10 million build steps have been logged, providing a rich source of build data.

Project X is rapidly evolving. Thus, to mitigate concept drift [27], we select a sample of 4,640 consecutive code and data builds from a recent 80-day period. During the period, an average of 58 build jobs were submitted daily, each taking an average of 21 minutes to execute. Each build in Project X is executed and logged in a stepwise fashion. On average, each build comprises 27

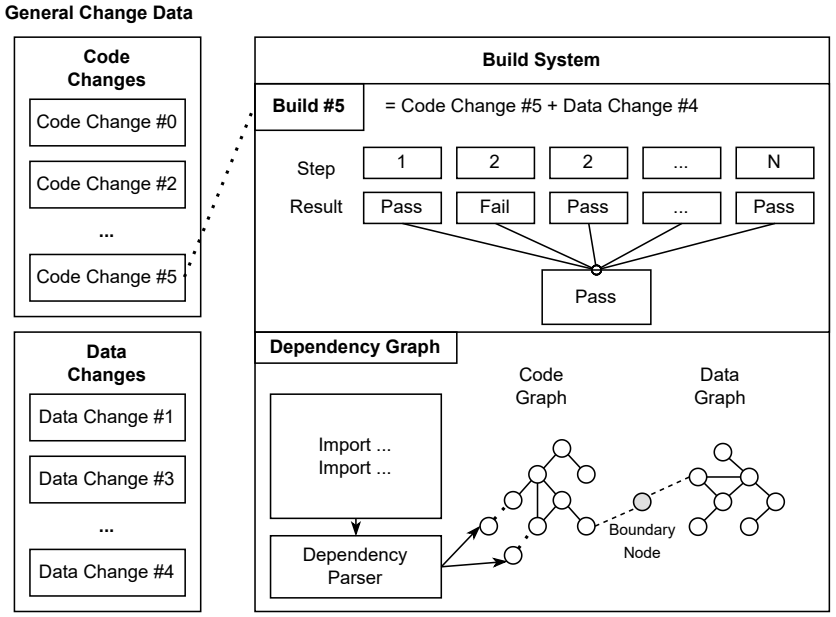

Fig. 1. Characteristics of Project X build system.

steps, with each step taking an average of 46 seconds to execute. In total, the code and data build jobs on the main branch of Project X consume 21 build hours daily. This excludes builds that are performed on other branches in Project X and the other games being developed by Ubisoft. With this in mind, it becomes clear why opportunities to reduce the time-to-feedback of the project and organizational build workload are being actively explored by Ubisoft.

General Change Data: Every change made to Project X is logged in Perforce and is indexed by a change number. The database includes typical change metadata, such as the author, timestamp, and updated file lists. Perforce provides the necessary data to compute process features for build outcome prediction, such as the number of lines of code added/edited/deleted, file revisions, committer identity, and file types.

Both code and data changes are stored in Perforce. At build time, a change to one discipline will be complemented by the most recent successfully built change from the other discipline. Figure 1 shows an example of the build process, where code change #5 triggers build #5 that is built using data change #4, which was included in passing build #4.

Build System: The build process of video games is particularly complex due to the size of the game project and the variety of file types that each must be transformed and assembled in a precise (and often unique) manner. Depending on the complexity of the change set, and the corresponding build, it may invoke hundreds of steps. To extract the build outcome, we first group the build steps by their change numbers and step names, as shown in Figure 1.

Builds can fail for many reasons. For example, network failures or bandwidth limits can introduce non-deterministic build failures even if the code and artifacts are not to blame. If a step fails, the Project X build system may automatically re-invoke it. Thus, the existence of a failing step does not necessarily indicate that the build outcome is a failure. Olewicki *et al.* [33] refer to these

non-deterministic build outcomes as "brown builds". To mitigate the noisy influence of brown builds, we label builds with failure outcomes only when the latest invocation of a build step that is associated with the build is failing. For example, Figure 1 shows that Build #5 has a passing outcome even though step 2 had an initially failing invocation (second column) because the step passed when it was retried (third column).

Multidisciplinary Dependency Graph: Video games are composed of a broad mix of digital artifacts in addition to source code [30], such as images, textures, and other so-called assets. These other digital artifacts are rapidly changing in tandem with the source code. Since their representations are often in binary form, traditional software engineering features like code churn cannot always be collected for change sets in this context. To better describe the data changes, our prior work [37] extracted the multidisciplinary dependency graph of Project X, which describes how all code and data artifacts are connected.

Similar to dependencies among code modules, other digital artifacts in video games also depend on each other in compositional and hierarchical ways. For example, a city in a game is composed of buildings, each of which being composed of building frames, doors, and windows, which may inherit properties like the texture weight and hardness of their composing materials. Therefore, dependency graphs can be constructed from data artifacts as well. In this way, dependency graphs of video games are multidisciplinary, *i.e.,* they combine the work products of software engineers (code), artists (textures, images), game designers (level design artifacts), and many more.

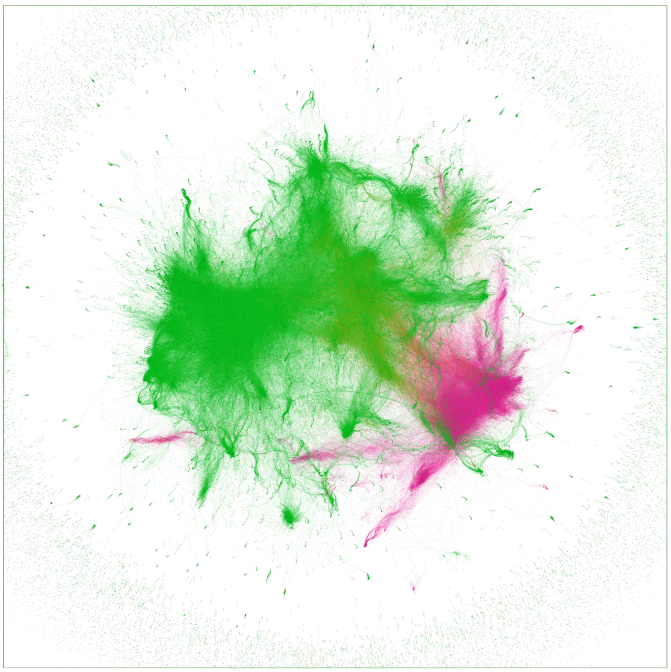

Fig. 2. Example of a file-level multidisciplinary dependency graph, with green nodes representing data nodes, pink nodes representing code nodes, and orange nodes representing boundary nodes. Extracting a multidisciplinary dependency graph of this scale requires extensive knowledge of Project X.

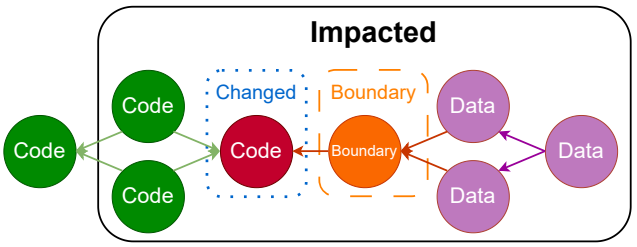

Fig. 3. Example of a video game dependency graph, with the boundary node lie between code and data nodes.

The complete dependency graph of Project X is not directly generated by the game engine (a critical component used in the development and build process of a video game). To construct a multidisciplinary dependency graph of Project X, we analyze the binary file that is generated by the game engine, which contains dependency information of the data artifacts, and the compile commands of the source codes, which contain the code dependency information. Therefore, each dependency graph of Project X consists of a code dependency sub-graph and a data dependency sub-graph. Figure 2 shows an example of the extracted multidisciplinary graph with millions of nodes and edges. Indeed, our analysis shows that 98.48% of the edges in our multidisciplinary graph are intra-node edges, *i.e.,* code-to-code or data-to-data edges. Figure 3 shows an example of the video game dependency graph with a clear boundary between the code and data dependencies. At the disciplinary boundaries, code and data nodes interact with each other through boundary nodes. Boundary nodes do not exist as files, but instead are generic and game-specific computational nodes that are provided by game developers for the artists to integrate their work into the game. Figure 1 shows an example of how the multidisciplinary dependency graph is constructed.

Although the game engine can produce the data dependency sub-graph and the compilation commands that contain the code sub-graph, recompiling all of the change sets that were recorded during the studied period would be impractical, as the number of data artifacts can reach the scale of millions. Figure 1 shows that our approach starts with an initial dependency graph and incrementally updates it using lightweight parsers that can extract dependency information from data artifacts and source files. For further details about the graph, please refer to our prior work [37].

3.2 Challenges of the Build Process in Project X

A close inspection of the Project X setting reveals two characteristics that hinder the application of build outcome prediction.

Challenge 1—Changes to data files often break the build: In video game development, data files are essential components that define the game's content, behavior, and appearance. The name "data file" is a general name for textures, models, animations, and other such game components. In Project X, there are more data files than source files (*e.g.,* `.cpp` and `.h`) in the repository. As an example, the latest multidisciplinary dependency graph in our dataset contains 1,104,037 nodes in total, while code nodes only account for 30,675 of them (2.8%).

In addition to their prevalence, data files in Project X are modified more frequently than code files. Indeed, of the 277 types of modified files, header files (`.h`) and source code (`.cpp`) rank as the 11^{th} and 14^{th} most frequently modified file types, respectively.

If builds are triggered by changes that only modify data files, they are referred to as data builds. Similar to builds that are triggered by code changes, data builds in video game development can also fail, since data builds also involve compiling the various data files into a format that the game

Table 1. Build failure rates and the number of impacted nodes of data and code changes. Data builds also incur build failures. Cross-boundary changes are more prone to build failures than other code changes.

	Total	Fail	Cross Boundary	Total	Fail	Mean Impact	Median Impact
Data	4,010	11%	N/A	N/A	N/A	452	2
Code	630	46%	Yes	278	51%	212,104	120,368
			No	352	41%	49	0[a]

[a] Impacted nodes counts exclude the directly changed nodes.

engine can efficiently use during gameplay as specified by their dependencies. Table 1 shows build failures for data builds account for 61% of all of the build failures in our dataset.

Data builds and their failures pose a challenge for traditional build outcome prediction approaches because prior build outcome features are mainly designed to represent code changes (*e.g.,* using the number of changed source code lines) that do not apply to data files. It is also impractical to design such features for data files because the definition would vary for each of the hundreds of types of data files that we observe in the Project X setting.

Challenge 2—Code changes with a cross-boundary impact break the build more often than code changes that have a code-only impact: In video game development, the interaction between code and data files defines the game functions, manages the game assets, and determines the player experience of the game. Cross-boundary changes occur when a code change updates a code file upon which data files depend. Figure 3 shows an example of a cross-boundary change using a sub-graph from a multidisciplinary dependency graph. The impacted nodes refer to the nodes that are dependent upon the changed node.

Table 1 shows that cross-boundary changes impact a large number of nodes in the dependency graph with a mean of 212,104 and a median of 120,368 nodes impacted, whereas data changes and changes that do not cross boundaries only impact a mean of 452 and 49 nodes, and a median of 2 and 0 nodes, respectively. The statistical significance of the discrepancies is confirmed by Mann-Whitney U tests (two-tailed, unpaired, $\alpha = 0.05$), yielding Holm-Bonferroni corrected p-values of 2.2×10^{-162} and 1.5×10^{-107}, respectively.

In addition to having a larger scale of impact, Table 1 also shows that 278 cross-boundary changes account for 44% of the code changes, with 51% of cross-boundary changes and 41% of the changes that do not cross boundaries being implicated in build failures. The statistical significance of this discrepancy is confirmed by Boschloo's Exact test, yielding a p-value of 0.008. We conclude that cross-boundary changes are significantly more likely to break the build pipeline than those changes that do not cross boundaries.

While cross-boundary changes do not (strictly speaking) present a challenge for the adoption of build outcome prediction in the Project X setting, they do present an opportunity that previous approaches have not yet exploited. Since changes that cross boundaries are significantly more likely to fail than changes that do not cross boundaries, we believe that leveraging this association will improve build outcome prediction in the Project X setting, and more broadly across Company Y.

In the Project X setting, 61% of build failures are incurred by data changes. Cross-boundary changes tend to impact a large number of nodes and are more prone to build failures than other code changes.

4 RAVENBUILD ON PROJECT X

In this section, we present a set of new features designed for build outcome prediction on Project X (Section 4.1), and an evaluation of the performance of the state-of-the-art BuildFast [3] model on Project X before and after adopting our features (Section 4.2).

4.1 Approach

We implement BuildFast [3]—the state-of-the-art build outcome prediction model that aims to provide early feedback—as our baseline to which RavenBuild can be compared. Indeed, other build outcome prediction models that aims to optimize CI cost-saving [22] are considered out of scope. More recent work has used deep learning to perform build outcome prediction [35, 36]; however, BuildFast and other state-of-the-art baselines were not compared. Moreover, the performance was not reported on the passing and failing classes separately. Prior work has shown that correctly predicting the failing builds is of greater practical concern than that of passing builds [22, 32].

Table 2 shows the features for build outcome prediction that have been adopted by BuildFast. In addition to adopting BuildFast features, we engineer features that are file type-agnostic to better accommodate the context of video game development into build outcome prediction. Table 3 provides an overview of RavenBuild features, which characterize three aspects of a change set.

Context-Aware Features: Change sets are typically submitted with the intention of integrating new features or bug fixes. This context transitively applies to the builds that are invoked to test a change set. We infer the context of a build from the metadata of its change set, such as commit messages, file types, and prior build activity. We classify change sets according to five intention categories using the keyword list that was originally proposed by Hindle *et al.* [20] and updated by Nayrolles and Hamou-Lhadj [31]. In addition to the intention of changes, we classify the studied changes based on the types of files being modified. Change sets can be code-only, data-only, or a mix of both. Finally, we implement a feature to detect if any of the commits in the change set being built have been built before. For example, in the Project X setting, builds can go through a preflight build before it is merged. In open-source settings, it is also possible for a commit to be built during the review of a pull request, and that same commit will be built again when the PR is merged.

Relevance-Aware Features: A set of builds may relate to a single task. For example, subsequent builds may be attempts to fix a failure that was introduced by an earlier build, or a large task may be decomposed into a series of incremental change sets that are built independently. Therefore, the features that exploit the status or context of a previous build should be constrained to the context when the prior and current builds are part of the same task. Thus, we propose features to suggest to the model when features that characterize previous builds are relevant for the current prediction, and when they should be ignored.

We compare the current build to its immediate predecessor to extract the relevance-aware features that are shown in Table 3. R_1, R_2, and R_4 are binary features, while R_3 is computed as $len(set(\textit{current actions}) - set(\textit{previous action}))$. Path relevance ($R_5$) measures the similarity between the modified directories of the previous and current builds. A pair of builds that modify files in the same location is more likely to build on interdependent targets than a pair of builds changing files in completely different locations. The path length is measured by counting the sub-directories in that common path. We then compute the relevance score as the length of the longest common path between the file paths of the previous (LCM_{prev}) and current builds (LCM_{cur}) divided by the sum of $len(LCM_{prev})$ and $len(LCM_{prev})$.

$$\textit{Path Relevance Score} = \frac{len(LCM_{prev,cur})}{len(LCM_{prev}) + len(LCM_{cur})} \tag{1}$$

Table 2. Build outcome features adopted by BuildFast.

BuildFast Features						
Current Build	C_1	src_churn	C_2	test_churn	C_3	src_ast_diff
	C_4	test_ast_diff	C_5	line_added	C_6	line_deleted
	C_7	files_added	C_8	files_deleted	C_9	files_modified
	C_{10}	src_files	C_{11}	test_files	C_{12}	config_files
	C_{13}	doc_files	C_{14}	class_changed	C_{15}	met_sig_modified
	C_{16}	met_body_modified	C_{17}	met_changed	C_{18}	field_changed
	C_{19}	import_changed	C_{20}	class_modified	C_{21}	class_added
	C_{22}	class_deleted	C_{23}	met_added	C_{24}	met_deleted
	C_{25}	field_modified	C_{26}	field_added	C_{27}	field_deleted
	C_{28}	import_added	C_{29}	import_deleted	C_{30}	commits
	C_{31}	fix_commits	C_{32}	merger_commits	C_{33}	committers
	C_{34}	by_core_member	C_{35}	is_master	C_{36}	time_interval
	C_{37}	day_of_week	C_{38}	time_of_day		
Previous Build	P_1	pr_state	P_2	pr_compile_error	P_3	pr_test_exception
	P_4	pr_tests_ok	P_5	pr_tests_fail	P_6	pr_duration
	P_7	pr_src_churn	P_8	pr_test_churn		
Historical Builds	H_1	fail_ratio_pr	H_2	fail_ratio_pr_inc	H_3	fail_ratio_re
	H_4	fail_ratio_com_pr	H_5	fail_ratio_com_re	H_6	last_fail_gap
	H_7	consec_fail_max	H_8	consec_fail_avg	H_9	consec_fail_sum
	H_{10}	commits_on_files	H_{11}	file_fail_prob_max	H_{12}	file_fail_prob_avg
	H_{13}	file_fail_prob_sum	H_{14}	pr_src_files	H_{15}	pr_src_files_in
	H_{16}	pr_test_files	H_{17}	pr_test_files_in	H_{18}	pr_config_files
	H_{19}	pr_config_files_in	H_{20}	pr_doc_files	H_{21}	pr_doc_files_in
	H_{22}	log_src_files	H_{23}	log_src_files_in	H_{24}	log_test_files
	H_{25}	log_test_files_in	H_{26}	team_size		

We compute the *sum* instead of the *max* because the longest common path of the build also provides a perspective on the dispersion across the changed files. A shorter *LCM* suggests that changed files are spread across multiple sub-directories, while a longer *LCM* suggests that the changed files are located in the same sub-directory. We strive to detect whether a low path relevance score is due to the broad dispersion of changed files, or due to changed files being located in completely different sub-directories, *e.g.,* consider two cases where in the first case, $len(LCM_{prev}) = 3$, $len(LCM_{cur}) = 4$, and $len(LCM_{prev,cur}) = 1$, whereas in the second case, $len(LCM_{prev}) = 3$, $len(LCM_{cur}) = 1$, and $len(LCM_{prev,cur}) = 1$. By using the *sum* in Equation 1, we can differentiate the two cases, whereas if we used the *max*, the path relevance score for both cases would be the same.

Dependency-Aware Features: Modifying files upon which a large number of files depend is riskier than modifying a file that has no dependents. For example, a change to a popular library function will propagate that change to its many use points, and is inherently more risky than changing a short script that calls a function from that library. Therefore, we propose a set of features that we extract from build dependency graphs as shown in Table 3. Our multidisciplinary approach to dependency graph construction (see Section 3) enables us to compute dependency-aware features on all types of files. We detect if the impact of a change set crosses the code-data boundary (*i.e.,* D_1), the total number of nodes impacted (*i.e.,* D_2), and the number of data/code/boundary nodes

Table 3. RavenBuild Features

	ID	Name	Description
Context	T_1	Is Corrective	If the commit message has 'fix', 'bug', 'wrong', 'fail', and 'problem'
	T_2	Is Additive	If the commit message has 'new', 'add', 'requirement', 'initial', and 'create'
	T_3	Is Non-Functional	If the commit message has 'doc' and 'merge'
	T_4	Is Perfective	If the commit message has 'clean' and 'better'
	T_5	Is Preventive	If the commit message has 'test', 'junit', 'coverage', and 'assert'
	T_6	Built-Before	If any of the commits in the current build have been built before (preflight)
	T_7	Change Type	The modified files are source files only, data files only, or a mix of both.
Relevance	R_1	Committer Changed	The pair of builds are submitted by different developers.
	R_2	Type Changed	The pair of builds have different change types (*i.e.,* T_7).
	R_3	Action Changed	The pair of builds have different actions.
	R_4	Is Same Intention	The pair of builds are of the same intention (*i.e.,* $T_1 - T_5$)
	R_5	Path Relevance	The similarity between the modified directories of the pair of builds
Dependency	D_1	Is Cross Boundary	Is a cross-boundary change
	D_2	#Impacted	Number of nodes impacted by the changed nodes
	D_3	#Impacted Data	Number of data nodes impacted by the changed nodes
	D_4	#Impacted Code	Number of code nodes impacted by the changed nodes
	D_5	#Impacted Boundary	Number of boundary nodes impacted by the changed nodes

impacted (*i.e.,* D_3–D_5). In the context of Project X, since data files also have dependencies, examining the set of impacted nodes provides an additional perspective on the riskiness of the change sets, which is not captured by any of the prior features.

Model Description: To study the contribution of the new features, we first adopt the original BuildFast model architecture and its hyper-parameters. BuildFast [3] uses an adaptive prediction mechanism that consists of two XGBoost [4] models: (1) one is trained on the previous builds that passed and current builds that failed, and (2) the other is trained on the previous builds that failed and current builds that also failed. During the training phase, the previous-passing model uses Information Gain [25] to select the top-25 features, whereas the previous-failing model uses Chi-Square Testing [13] to select the top-30 features. At inference time, BuildFast predicts build outcomes based on their immediate predecessor's outcome, *i.e.,* if the previous build failed, BuildFast predicts the current build outcome using the previous-failing model, whereas if the previous build passed, BuildFast predicts the current build outcome using the previous-passing model.

Model Evaluation: To respect the chronological order of builds, the most recent one-third of our dataset is held out as the testing set, with the prior two-thirds being used for training. Table 4 shows the performance of BuildFast and RavenBuild, and the difference between the two in F1-fail, F1-pass, recall-fail, recall-pass, precision-fail, precision-pass, and AUC when applied to Project X. We also compute the benefit, cost, and gain metrics that were proposed by Chen *et al.* [3]. Benefit is the number of build hours that would have been saved by skipping the correctly predicted passing builds. Cost is the number of unnecessary build hours that would have been spent due to incorrectly predicted failing builds. Gain is the difference between benefit and cost, *i.e., Benefit − Cost.*

4.2 Results

Observation 1: In Project X, RavenBuild matches or outperforms BuildFast in terms of F1-score, recall, precision, and AUC, but underperforms in terms of precision-fail and recall-pass. Table 4 shows that after adopting the context-, relevance-, and dependency-aware features, RavenBuild outperforms BuildFast in most of the failing classes, by 21.2 percentage points of F1-fail and 31.1 percentage points of recall-fail. Although RavenBuild also incurs a 1.9 percentage point

Table 4. Performances of BuildFast, RavenBuild, and the improvements on Project X

	BuildFast	RavenBuild	Improvement
F1-fail	.426	.638	+.212
F1-pass	.963	.967	+.004
Recall-fail	.296	.607	+.311
Recall-pass	.991	.972	-.019
Precision-fail	.755	.672	-.083
Precision-pass	.936	.963	+.027
AUC	.791	.877	+.086
Benefit	301.3	280.4	-20.9
Cost	83.5	40.3	+43.2
Gain	217.8	240.1	+22.3

Table 5. Open-Source projects and Project X Statistics

ID	Project	Duration[a]	Pass	Fail	ID	Project Name	Duration[a]	Pass	Fail
1	atlasdb	37	2,135	1,021	12	interlok	7	120	29
2	auth0-java	2.3	328	29	13	java-jwt	3	274	55
3	cassandra	117	313	180	14	micrometer	18	322	76
4	cbioportal	29	58	217	15	opennms [b]	1	244	37
5	client_java	4	87	18	16	react-native-share	12	24	41
6	conjure-java-runtime	7	354	83	17	rskj	18	844	198
7	docker-compose-rule	4	105	131	18	spring-cloud-aws	5	419	265
8	fresco	11	79	559	19	spring-cloud-security	2	113	366
9	gradle-baseline	8	378	111	20	Strata	16	136	27
10	gradle-git-version	6	96	37	21	styx	9	109	81
11	grakn	20	106	13	22	testcontainers-java	2	1018	16
X	Project X	21	3,898	743					

[a]in minutes. [b]opennms-provisioning-integration-server.

penalty in recall-pass and 8.3 percentage point penalty in precision-fail, RavenBuild outperforms BuildFast by 0.4 percentage points in F1-pass, 2.7 percentage points in precision-pass, and 8.6 percentage points in AUC.

Observation 2: Although RavenBuild does not save as much as BuildFast, it makes fewer mistakes when recommending builds, leading to a better overall gain score. Table 4 shows that RavenBuild accrues 20.9 fewer build hours of savings when compared to BuildFast. We suspect that this is because RavenBuild has a slightly lower recall-pass score. On the other hand, RavenBuild reduces costs by 40.3 build hours by more accurately predicting the failing class. The overall perspective shows that RavenBuild improves the net gain of build outcome prediction in the Project X setting by 22.3 build hours.

RavenBuild outperforms the state-of-the-art approach to build outcome prediction in the Project X setting by a substantial amount.

5 RAVENBUILD-CR ON OPEN-SOURCE PROJECTS AND PROJECT X

While dependency-aware features are promising a new direction for build outcome prediction, the dependency graphs from which they are extracted are expensive to construct and require project-specific knowledge to be reliably produced. Dependency features can be generalized to open-source projects with few data artifacts. Features D_2–D_5 compute the number of files that are affected by the change set through dependencies. The challenging aspect is the project-specific knowledge required to define disciplinary boundaries for different types of projects. Software repositories may contain data artifacts like pre-trained models (*e.g.,* `.tflite`), front-end design (*e.g.,* `.html`), or configuration files (*e.g.,* `.xml`, `.pom`, `.config`) and others may also have more than one type of programming language. Project-specific knowledge is also required to obtain dependency graphs for each submitted change. In settings that have many projects, the effort required to produce the dependency features quickly becomes impractical. Therefore, in this section, we present an alternative RavenBuild-CR that only uses the context- and relevance-aware features and excludes dependency-aware features (Section 5.1). Since the need for deep knowledge of dependency structures has been relaxed, we evaluate RavenBuild-CR model on Project X and on a sample of active open-source projects (Section 5.2).

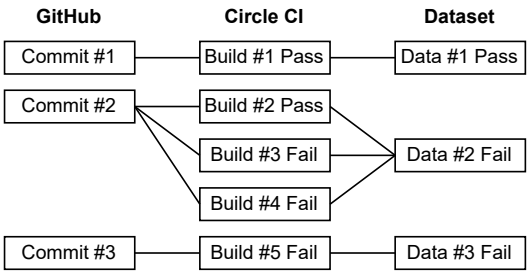

Fig. 4. Builds triggered on the same commit ID are merged into one data point.

5.1 Approach

Studied Open-Source Projects: Gallaba *et al.* [10] provide a large dataset that contains 23,330,690 builds that were performed using Circle CI [5] that span 7,795 projects and 40 programming languages. To be consistent with BuildFast, we exclude all repositories for which Java is not the primary language (*i.e.,* GitHub uses the Linguist software[1] to determine the primary language). Then, for each such project, we collect its build data from Circle CI. Since Circle CI does not retain data about all of the past builds, we filter out the repositories that have fewer than 100 builds available at the time of our analysis, resulting in a dataset with 45,247 builds from 22 projects.

Circle CI is a versatile platform. As such, the studied projects configure the platform in various ways to meet their needs. For example, we observe that some projects trigger multiple builds with different parameter settings on the same change set to, *e.g.,* execute test cases in parallel. Since many features are extracted at the change set level, we avoid duplicate entries in our dataset by merging builds that are triggered for the same commit ID as shown in Figure 4. We consider that the merged entry has a passing build outcome if and only if all of the merged entries have passing outcomes. The duration of the merged build is calculated by summing up the durations of each of the merged entries. Finally, to ensure that previous build features can be computed, we exclude

[1] https://github.com/github-linguist/linguist/

Table 6. Performances of BuildFast and RavenBuild-CR on Open-Source projects and Project X.

	Open-Source			Project X		
	BuildFast	RavenBuild-CR	Improvement	BuildFast	RavenBuild-CR	Improvement
F1-fail	.478	.502	+.024	.426	.555	+.129
F1-pass	.756	.771	+.015	.963	.968	+.005
Recall-fail	.583	.612	+.029	.296	.430	+.134
Recall-pass	.732	.736	+.004	.991	.989	-.002
Precision-fail	.457	.489	+.032	.755	.784	+.029
Precision-pass	.805	.842	+.037	.936	.948	+.012
AUC	.723	.738	+.015	.791	.861	+.070
Benefit	331.8	356.9	+25.1	301.3	300.8	-.5
Cost	207.0	200.2	+6.8	83.5	61.1	+22.4
Gain	124.8	156.7	+31.9	217.8	239.7	+21.9

the first build of each project, since it will not have a predecessor from which the features can be extracted. Table 5 provides an overview of the 22 studied open-source projects and Project X.

5.2 Results

Observation 3: With only the context-aware and relevance-aware features, RavenBuild-CR still outperforms BuildFast on Project X. Without the dependency-aware features, Table 6 shows that RavenBuild-CR still outperforms BuildFast in the Project X setting by 12.9 percentage points in F1-fail, 13.4 percentage points in recall-fail, 2.9 percentage points in precision-fail, and 7 percentage points in AUC. RavenBuild-CR also improves slightly on the performance in the passing class as well, even outperforming RavenBuild itself in terms of precision-pass (cf. Table 4).

In terms of time-to-feedback, RavenBuild-CR leads to 22.4 hours less in Cost, with a negligible 0.5 hours reduction in Benefit, resulting in 21.9 more hours of Gain than BuildFast. Although RavenBuild-CR improves both precision and recall on Project X, the improvements in the passing class are relatively small. Since the benefit is calculated by summing up the execution time of the correctly predicted passing builds, we suspect that the minor decrease in benefit is due to the build duration of the correctly predicted passing builds of RavenBuild-CR being slightly shorter than the passing builds that are correctly predicted by BuildFast.

Observation 4: RavenBuild-CR outperforms BuildFast on all metrics in the open-source setting. Table 6 also shows that RavenBuild-CR outperforms BuildFast in the open-source setting by 2.4 percentage points in F1-fail, 1.5 percentage points in F1-pass, 2.9 percentage points in recall-fail, 0.4 percentage points in recall-pass, 3.2 percentage points in precision-fail, 3.7 percentage points in precision-pass, and 1.5 percentage points in AUC. In terms of time-to-feedback, RavenBuild-CR saves a total of 25.1 hours more than BuildFast. As RavenBuild-CR also incurs 6.8 hours less Cost than BuildFast, the net Gain is improved by 31.9 hours.

Without dependency-aware features, although a negligible reduction in Benefit was observed, RavenBuild-CR still outperforms the state of the art in the Project X setting. Moreover, RavenBuild-CR is general enough to apply to the open-source setting, where we observe across-the-board improvements with respect to the state of the art.

Table 7. The ablated performance of RavenBuild with various families of features disabled.

	Open-Source			Project X			
	RavenBuild-C	RavenBuild-R	RavenBuild-CR	RavenBuild-C	RavenBuild-R	RavenBuild-D	RavenBuild
F1-fail	.489	.481	**.502**	.560	.45	.571	**.638**
F1-pass	**.775**	.774	.771	.968	.962	**.968**	.967
Recall-fail	.598	.583	**.612**	.430	.333	.459	**.607**
Recall-pass	.741	**.743**	.736	**.990**	.986	.986	.972
Precision-fail	.467	.462	**.489**	**.806**	.692	.756	.672
Precision-pass	.839	.839	**.842**	.948	.939	.95	**.963**
AUC	**.744**	.728	.738	.818	.823	.797	**.877**
Benefit	350.8	334.2	**356.9**	**297.1**	294.4	289.7	280.4
Cost	**186.6**	198.3	200.2	58.8	77.3	56.7	**40.3**
Gain	**164.2**	136	156.7	238.3	217.1	232.2	**240.1**

6 ABLATION STUDY

In this section, we conduct an ablation study to understand the contribution that the context-, relevance-, and dependency-aware features make to RavenBuild.

6.1 Approach

To assess the importance of each family of features, we train and evaluate RavenBuild with various combinations of features disabled, and observe the model performance. Table 7 shows the result of our ablation study to measure the contribution of the context-aware features (*i.e.,* RavenBuild-C), relevance-aware features (*i.e.,* RavenBuild-R), and dependency-aware features (*i.e.,* RavenBuild-D) to RavenBuild in the open-source projects and Project X.

6.2 Results

Observation 5: RavenBuild-CR outperforms RavenBuild-C and RavenBuild-R in F1-fail, recall-fail, precision-fail, and precision-pass. With only the context-aware features, RavenBuild-C is outperformed by RavenBuild-CR in the open-source projects by 1.3 percentage points in F1-fail, 1.4 percentage points in recall-fail, 2.2 percentage points in precision-fail, and 0.3 percentage points in precision-pass. With only the relevance-aware features, RavenBuild-R is outperformed by RavenBuild-CR in the open-source projects by 2.1 percentage points in F1-fail, 2.9 percentage points in recall-fail, 2.7 percentage points in precision-fail, and 0.3 percentage points in precision-pass. RavenBuild-C or RavenBuild-R only negligibly outperforms RavenBuild-CR by 0.4 percentage points in F1-pass, 0.7 percentage points in recall-pass, and 0.6 percentage points in AUC.

Observation 6: RavenBuild outperforms RavenBuild-C, RavenBuild-R, and RavenBuild-D in F1-fail, recall-fail, precision-pass, and AUC. With only the context-aware features, RavenBuild-C is outperformed by RavenBuild in Project X by 7.8 percentage points in F1-fail, 17.7 percentage points in recall-fail, 0.3 percentage points in precision-pass, and 5.9 percentage points in AUC. With only the relevance-aware features, RavenBuild-R is outperformed by RavenBuild in Project X by 18.8 percentage points in F1-fail, 27.4 percentage points in recall-fail, 1.5 percentage points in precision-pass, and 5.4 percentage points in AUC. With only the dependency-aware features, RavenBuild-R is outperformed by RavenBuild in Project X by 6.7 percentage points in F1-fail, 14.8 percentage points in recall-fail, 1.3 percentage points in precision-pass, and 8 percentage points in AUC. RavenBuild-C, RavenBuild-R, or RavenBuild-D only negligibly outperforms RavenBuild by 0.1 percentage points in F1-pass and 1.8 percentage points in recall-pass. Although RavenBuild-C performs the best among all model variations in precision-fail, RavenBuild has the best overall performance for its superiority in F1-fail, recall-fail, and AUC.

Table 8. The performance of Parrot, BuildFast, and RavenBuild. Values in boldface fonts indicate the best performance among the three.

	Open-Source			Project X		
	Parrot	BuildFast	RavenBuild-CR	Parrot	BuildFast	RavenBuild
F1-fail	.421	.478	**.502**	**.657**	.426	.638
F1-pass	**.797**	.756	.771	**.967**	.963	**.967**
Recall-fail	.422	.583	**.612**	**.652**	.296	.607
Recall-pass	**.797**	.732	.736	.968	**.991**	.972
Precision-fail	.420	.457	**.489**	.662	**.755**	.672
Precision-pass	.798	.805	**.842**	**.967**	.936	.963
AUC	.610	.723	**.738**	.810	.791	**.877**
Benefit	336.3	331.8	**356.9**	292.7	**301.3**	280.4
Cost	**158.3**	207.0	200.2	**39.1**	83.5	40.3
Gain	**178**	124.8	156.7	**253.6**	217.8	240.1

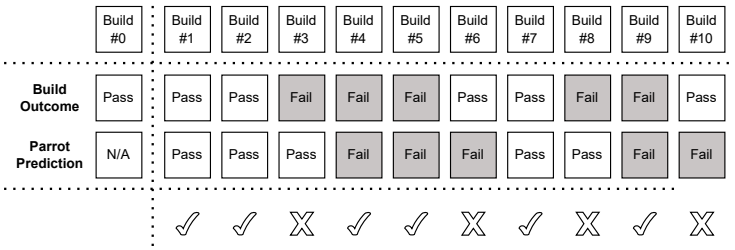

Fig. 5. The parrot approach repeats the previous build outcome as its prediction, achieving accuracy, precision, and recall of .60.

Context-, relevance-, and dependency-aware features all substantially contribute to the performance of RavenBuild in the open-source projects and Project X.

7 AN INTROSPECTIVE LOOK AT THE STATE OF BUILD OUTCOME PREDICTION

Although RavenBuild and RavenBuild-CR improve on the state-of-the-art BuildFast model, we find that a naïve "parrot" approach to build outcome prediction that simply echoes the previous build as a prediction for the current one can perform as well as these state-of-the-art approaches. In this section, we describe the parrot approach (Section 7.1) and evaluate the performance of Parrot in comparison to BuildFast, RavenBuild, and RavenBuild-CR (Section 7.2).

7.1 Approach

Figure 5 provides an example application of Parrot. We use Parrot as a benchmark to which we compare the performance of BuildFast, and RavenBuild-CR on the studied open-source projects, and BuildFast and RavenBuild on Project X.

Figure 6 plots the performance of the parrot approach (y-axis) against that of BuildFast and RavenBuild/RavenBuild-CR (x-axis) in terms of precision-fail, precision-pass, recall-fail, and recall-pass, respectively. Each open-source project is represented by a point, and Project X is denoted by

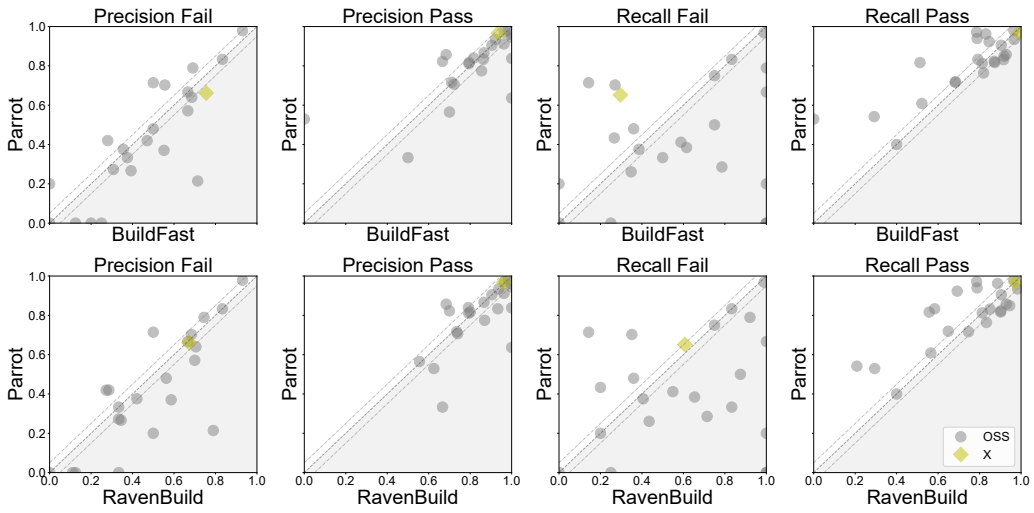

Fig. 6. RavenBuild (Project X), RavenBuild-CR, and BuildFast cannot consistently outperform the parrot approach.

a diamond-shaped point. To visually compare the performance of BuildFast and RavenBuild with the Parrot approach, we divide the coordinate space with a diagonal line into unshaded and shaded regions. Points falling in the unshaded region indicate that the Parrot approach yields better results, whereas points falling in the shaded region indicate that the other approach is performing better.

To further refine the comparison, we include two lines adjacent to the diagonal line, which denote a 5% confidence interval. We consider points that fall within the confidence interval to have a performance within the margin of error, and the difference between the model and Parrot is likely indistinguishable. Conversely, points that fall outside of the confidence interval are considered to have performance that is distinguishable between the state-of-the-art models and the parrot approach. Therefore, by counting the number of points falling in the unshaded region, we can infer how often Parrot outperforms state-of-the-art build outcome prediction models.

To study why naïve predictions that simply echo the previous build outcome achieve comparable performance to the state of the art, we count the number of builds that have the same outcome as their immediate predecessors. Table 9 shows the total number of passing and failing builds, the number of passing and failing builds that have the same outcome as their immediate predecessors, and the relative percentages in each category for the studied open-source projects and Project X.

7.2 Results

Observation 7: The naïve Parrot outperforms both BuildFast and RavenBuild in terms of time-to-feedback. Table 8 shows the performance of Parrot, BuildFast, and RavenBuild(-CR). While both BuildFast and RavenBuild(-CR) outperform Parrot in F1-fail, recall-fail, precision-fail, precision-pass and AUC in the open-source context, and in F1-pass, recall-pass, precision-fail, and AUC in the Project X context, Parrot outperforms both approaches in terms of Gain by reducing the Cost considerably.

Observation 8: BuildFast, RavenBuild, and RavenBuild-CR often underperform with respect to Parrot in terms of precision and recall. Figure 6 shows 7, 10, 6, and 12 points in the four sub-graphs associated with BuildFast performance, and 6, 8, 5, and 11 points in the four sub-graphs

Table 9. The number of builds that have the same outcome as their immediate predecessor in the passing and failing class, the total number of builds in the passing and failing class, and the percentage of the builds that have the same build outcome as their immediate predecessor in all builds in the passing and failing class.

	Open-Source			Project X		
	Passing	Failing	All	Passing	Failing	All
Same[a]	6,302	2,229	8,531	3,587	744	4,031
Total	7,662	3,590	11,252	3,898	743	4,640
%	82%	62%	76%	92%	59%	82%

[a] Builds that have same outcomes as their immediate predecessor.

associated with RavenBuild performance, indicating that the parrot approach can outperform BuildFast and RavenBuild in at least one of the evaluation metrics (*i.e.,* precision-fail, precision-pass, recall-fail, and recall-pass) in at least seven of the studied projects. Overall, BuildFast and RavenBuild underperform with respect to the parrot approach in 38% and 33% of the 92 evaluation cases (*i.e.,* 23 projects × 4 evaluation metrics). Indeed, a project-level inspection reveals that although Parrot does not outperform BuildFast and RavenBuild in terms of all of the studied evaluation metrics, Parrot outperforms BuildFast and RavenBuild in, respectively, 6 and 4 of the 23 studied projects in terms of 3 of the 4 evaluation metrics.

In the cases where build outcome prediction models outperform the Parrot approach, the difference in performance is often within the margin of error. Figure 6 shows BuildFast and RavenBuild only outperform the Parrot approach by more than the margin of error in 5–12 of the 23 studied projects in terms of precision-fail, precision-pass, recall-fail, and recall-pass. In 64% and 61% of the evaluated cases, BuildFast and RavenBuild do not significantly outperform the Parrot approach.

We suspect that Parrot takes advantage of repeated build outcomes and provides predictions that offer little practical value. For example, if the testing set contains *True* labels only, a naïve model can achieve perfect performance in all evaluation metrics by simply predicting *True*, whereas it would be much harder for any model to score 100% in all evaluation metrics. Although Parrot outperforms both BuildFast and RavenBuild, Parrot fails to predict the cases when a build outcome flips from passing to failing or vice versa. These cases are the ones that provide developers with the most insight. Therefore, Parrot outperforming BuildFast and RavenBuild does not imply the build outcome prediction work is meaningless. Instead, Parrot should be interpreted as a baseline that encompasses the degree to which sequences of outcomes are present in the dataset.

Observation 9: Build outcomes tend to repeat their previous outcome. The fact that the Parrot approach achieves performance that is comparable to or better than the state-of-the-art models reveals build outcomes tend to repeat their previous outcome. Indeed, Table 9 shows that 76% of builds from open-source projects have the same outcome as their previous builds. More specifically, 82% of the passing builds and 62% of the failing builds have the same outcome as their immediately preceding build. In terms of Project X, 87% of all builds, 92% of the passing builds, and 59% of the failing builds have the same outcome as their immediately preceding build. Our finding aligns with the observation of Jin and Servant [22], and demonstrates that a bias towards the previous build outcome exists not only in open-source contexts but also in AAA video game projects like Project X.

Despite the improvements achieved in build outcome prediction, both RavenBuild and BuildFast are often outperformed by the naïve Parrot approach. However, Parrot cannot capture the flip cases, which are of greatest practical importance. Nonetheless, we recommend that future work on build outcome prediction include Parrot performance as a tendency indicator to ground the evaluation.

8 THREATS TO VALIDITY

Below, we describe the threats to the validity of our study.

8.1 Construct Validity

Construct validity concerns may creep into our study if our measurements are misaligned with the phenomena that we set out to study. In our work, these threats could be related to the selection of evaluation metrics. We rely on cost metrics from state-of-the-art studies [3] to ensure a fair comparison between RavenBuild, BuildFast [3], and other baseline approaches. Nevertheless, the cost measured by these metrics might still not reflect the real cost incurred by build outcome prediction in practice. More specifically, in deployments that aggressively pursue savings, negative side effects may emerge when these approaches are actually adopted. To capture such costs, we would need to deploy the studied approaches in a live CI service and measure costs that are actually impacting practitioners and stakeholders. Unfortunately, this evaluation is not feasible at this stage given the critical role that CI services play in production pipelines at Ubisoft.

8.2 Internal Validity

One potential threat to our internal validity is the existence of flaky builds in the studied datasets. Flaky builds are builds that fail intermittently due to non-deterministic factors in the system under test, the CI service, or the deployment environment. At Ubisoft, failing build steps are automatically retried. Therefore, to mitigate the effect of flaky builds on our study and datasets, we consider a build step to be passing as long as it has one passing execution.

Another threat to internal validity has to do with the use of machine learning in the studied approaches, especially the steps of feature selection in the BuildFast model architecture. Some features might be left out in the feature selection process. Through project-level inspection, we confirm that the RavenBuild features are selected to train the models in the feature selection step.

In addition, the dependency-aware features are collected from the multidisciplinary dependency graphs, which take time to be fully extracted. To address the issue, we propose to use the previous version of the dependency graph. Indeed, our analysis shows that only 34% of the studied changes alter edges within the graph, but those changes only affect one side of the dependency (*i.e.,* the files upon which the added files depend), and thus would not invalidate dependency analyses performed on the prior graph state. The changes that do alter the graph structure in analysis-invalidating ways are easily identified by scanning for differences in the `import` section of the edited files, which can be used to prevent results of the dependency analysis that are incorrect from being adopted by the model. However, future work needs to address the cases where the prior dependency graph is invalidated for analysis.

8.3 External Validity

Threats to external validity have to do with the generalizability of our results to other systems. To ensure consistency with BuildFast, we sample Java projects from GitHub to construct our open-source dataset. The performance of RavenBuild might differ when applied to other projects, but the key insights of the context-, relevance-, and dependency-aware features remain the same,

as these features are extracted from change information that does not depend on the platform or programming languages.

8.4 Reliability Validity

Reliability validity threats may impact the replicability of this study. Due to legal constraints, we cannot share our dataset from Project X, since datasets collected on a proprietary project may leak confidential information. We conduct our evaluation of state-of-the-art approaches on 22 open-source projects as well as Project X. This selection of the same language of subject systems as prior work [3] ensures consistency when comparing results.

9 CONCLUSION

In this paper, we describe the unique characteristics of AAA video game project development that make build outcome prediction particularly challenging. In the context of Project X, we observe that data changes that do not modify source code also incur build failures, as data artifacts should be compiled in an order-sensitive manner that respects the specified dependencies. Moreover, code changes that have an impact that crosses the source-data boundary are more prone to build failures than code changes that do not impact data files. The method by which we construct the multidisciplinary dependency graph forges a new direction for the extraction of features for build outcome prediction. To better accommodate the data changes into build outcome prediction, and to exploit the abundant and rich dependency data, we propose features that are context, relevance, and dependency aware. We incorporate those features into RavenBuild—a novel solution for build outcome prediction. We find that RavenBuild outperforms BuildFast by a substantial amount on Project X. While extracting the dependency graphs requires extensive project-specific understanding, we provided an alternative to RavenBuild that only adopts the context-aware and relevance-aware features, namely RavenBuild-CR. We find that RavenBuild-CR outperforms BuildFast in both the open-source and Project X settings. Surprisingly, however, we find that neither BuildFast nor RavenBuild can consistently outperform a naïve Parrot approach that simply echoes the previous build outcome as predictions. Though Parrot fails to capture the important flip cases, the comparable performance of Parrot reveals that build outcomes tend to repeat their immediate predecessor. Therefore, future build outcome prediction work should consider Parrot performance as a tendency indicator when evaluating.

While the concept of multidisciplinary dependency graph and cross-boundary changes can also generalize to other settings, the project-specific knowledge required to construct the dependency graph for every submitted change set constrains researchers from conducting dependency analysis at a large-scale across projects. Therefore, we encourage future replication work to be conducted to further demonstrate the effectiveness of dependency-aware features in build outcome prediction.

10 DATA AVAILABILITY

The presented dataset and results, as well as data collection scripts for open-source projects, are available in our replication package.[2]

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of MITACS Canada.

[2] https://doi.org/10.5281/zenodo.10719029

REFERENCES

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. 2021. A Machine Learning Approach to Improve the Detection of CI Skip Commits. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2740–2754. https://doi.org/10.1109/TSE.2020.2967380

[2] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. 2021. Which Commits Can Be CI Skipped? *IEEE Transactions on Software Engineering* 47, 3 (2021), 448–463. https://doi.org/10.1109/TSE.2019.2897300

[3] Bihuan Chen, Linlin Chen, Chen Zhang, and Xin Peng. 2020. BUILDFAST: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 42–53.

[4] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[5] CircleCI. 2023. *CircleCI*. https://circleci.com/ Accessed on Date,September 26, 2023.

[6] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.

[7] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 235–245. https://doi.org/10.1145/2635868.2635910

[8] Daniel Elsner, Roland Wuersching, Markus Schnappinger, Alexander Pretschner, Maria Graber, René Dammer, and Silke Reimer. 2022. Build System Aware Multi-language Regression Test Selection in Continuous Integration. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 87–96. https://doi.org/10.1145/3510457.3513078

[9] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh. 2022. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions on Software Engineering* 48, 6 (2022), 2040–2052. https://doi.org/10.1109/TSE.2020.3048335

[10] Keheliya Gallaba, Maxime Lamothe, and Shane McIntosh. 2022. Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI. In *Proc. of the International Conference on Software Engineering (ICSE)*. 1330–1342.

[11] Keheliya Gallaba and Shane McIntosh. 2020. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects That (Mis)Use Travis CI. *IEEE Transactions on Software Engineering* 46, 1 (2020), 33–50. https://doi.org/10.1109/TSE.2018.2838131

[12] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 211–222. https://doi.org/10.1145/2771783.2771784

[13] Priscilla E Greenwood and Michael S Nikulin. 1996. *A guide to chi-squared testing*. Vol. 280. John Wiley & Sons.

[14] Ahmed E. Hassan and Ken Zhang. 2006. Using Decision Trees to Predict the Certification Result of a Build. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*. IEEE Computer Society, 189–198. https://doi.org/10.1109/ASE.2006.72

[15] Foyzul Hassan and Xiaoyin Wang. 2017. Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 157–162. https://doi.org/10.1109/ESEM.2017.23

[16] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The Art of Testing Less without Sacrificing Quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 483–493.

[17] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 197–207. https://doi.org/10.1145/3106237.3106270

[18] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. (2016), 426–437. https://doi.org/10.1145/2970276.2970358

[19] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358

[20] Abram Hindle, Daniel M. German, and Ric Holt. 2008. What Do Large Commits Tell Us? A Taxonomical Study of Large Commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories* (Leipzig, Germany) *(MSR '08)*. Association for Computing Machinery, New York, NY, USA, 99–108. https://doi.org/10.1145/1370750.1370773

[21] Xianhao Jin. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1650–1654. https://doi.org/10.1145/3468264.3473103

[22] Xianhao Jin and Francisco Servant. 2020. A Cost-Efficient Approach to Building in Continuous Integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 13–25. https://doi.org/10.1145/3377811.3380437

[23] Xianhao Jin and Francisco Servant. 2022. HybridCISave: A Combined Build and Test Selection Approach in Continuous Integration. *ACM Trans. Softw. Eng. Methodol.* (dec 2022). https://doi.org/10.1145/3576038 Just Accepted.

[24] Xianhao Jin and Francisco Servant. 2022. Which Builds Are Really Safe to Skip? Maximizing Failure Observation for Build Selection in Continuous Integration. *J. Syst. Softw.* 188, C (jun 2022), 18 pages. https://doi.org/10.1016/j.jss.2022.111292

[25] Changki Lee and Gary Geunbae Lee. 2006. Information gain and divergence-based feature selection for machine learning-based text categorization. *Information Processing & Management* 42, 1 (2006), 155–165. https://doi.org/10.1016/j.ipm.2004.08.006 Formal Methods for Information Retrieval.

[26] Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V. Mäntylä, and Tomi Männistö. 2015. The highways and country roads to continuous deployment. *IEEE Software* 32, 2 (2015), 64–72. https://doi.org/10.1109/MS.2015.50

[27] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. 2018. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering* 31, 12 (2018), 2346–2363.

[28] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. 2011. An Empirical Study of Build Maintenance Effort. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 141–150. https://doi.org/10.1145/1985793.1985813

[29] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16

[30] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. 2014. Cowboys, Ankle Sprains, and Keepers of Quality: How is Video Game Development Different from Software Development?. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2568225.2568226

[31] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. CLEVER: Combining Code Metrics with Clone Detection for Just-in-Time Fault Prevention and Resolution in Large Industrial Projects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 153–164.

[32] Ansong Ni and Ming Li. 2017. Cost-Effective Build Outcome Prediction Using Cascaded Classifiers. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 455–458. https://doi.org/10.1109/MSR.2017.26

[33] Doriane Olewicki, Mathieu Nayrolles, and Bram Adams. 2022. Towards Language-Independent Brown Build Detection. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2177–2188. https://doi.org/10.1145/3510003.3510122

[34] Cong Pan and Michael Pradel. 2021. Continuous Test Suite Failure Prediction. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 553–565. https://doi.org/10.1145/3460319.3464840

[35] Islem Saidani, Ali Ouni, Moataz Chouchen, and Mohamed Wiem Mkaouer. 2021. BF-Detector: An Automated Tool for CI Build Failure Detection. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1530–1534. https://doi.org/10.1145/3468264.3473115

[36] Islem Saidani, Ali Ouni, and Mohamed Wiem Mkaouer. 2022. Improving the Prediction of Continuous Integration Build Failures Using Deep Learning. *Automated Software Engg.* 29, 1 (may 2022), 61 pages. https://doi.org/10.1007/s10515-021-00319-5

[37] Gengyi Sun, Mehran Meidani, Sarra Habchi, Mathieu Nayrolles, and Shane McIntosh. 2024. Code Impact Beyond Disciplinary Boundaries: Constructing a Multidisciplinary Dependency Graph and Analyzing Cross-Boundary Impact. In *Proc. of the International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3639477.3639726

[38] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations*

of Software Engineering (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 805–816. https://doi.org/10.1145/2786805.2786850

[39] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2012. FaultTracer: A Change Impact and Regression Fault Analysis Tool for Evolving Java Programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 40, 4 pages. https://doi.org/10.1145/2393596.2393642

[40] Thomas Zimmermann and Nachiappan Nagappan. 2007. Predicting Subsystem Failures using Dependency Graph Complexities. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*. 227–236. https://doi.org/10.1109/ISSRE.2007.19

[41] Manuela Züger and Thomas Fritz. 2015. Interruptibility of Software Developers and Its Prediction Using Psycho-Physiological Sensors. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY, USA, 2981–2990. https://doi.org/10.1145/2702123.2702593

Received 2023-09-29; accepted 2024-01-23