APPENDIX
Continuous Integration and Software Quality: A Causal Explanatory Study

1st Omitted for Submission
Omitted Institution
Omitted Institution
Omitted, Ommited
omitted@ommitted.com

2nd Omitted for Submission
Omitted Institution
Omitted, Ommited
omitted@ommitted.com

3rd Omitted for Submission
Omitted Institution
Omitted, Ommited
omitted@ommitted.com

I. METHODOLOGY

We follow a pipeline in 5 stages presented in Fig. 1 to answer our research questions: The Literature Review (stage 1) and DAG Building (stage 2) that contributes to answering RQ1. The Mining Software Repository (stage 3) and the DAG Implications Testing (stage 4) address the RQ2. Lastly, another stage of DAG Building (stage 5) iterates with the DAG Implications Testing stage (stage 4) until a DAG is obtained with literature and data consistency to answer RQ3.

A. *What does the literature proclaim about CI and software quality?*

As depicted in Fig. 1, we perform two research stages to answer RQ1. First, the literature review compiles the existing assumptions about how CI may impact software quality and allow us to map variables, associations, and every common cause for any two or more variables in the DAG. This mapped information is an input to stage 2 - DAG building. We execute these two stages iteratively to build a causally sufficient DAG representing the literature knowledge.

*1) **Literature Review**:* To review the relationship between continuous integration (CI) and software quality (SQ), we consider "issues", "bugs", and/or "defects" as proxies of software quality. Since we are also interested in knowing the common causes between CI and these variables (i.e., 'issues", "bugs", or "defects"), we review the literature for each one of these variables.

We also review the literature for each new variable found during the DAG building in an iterative approach. We perform the searches on Google Scholar[1] using the name of variables or synonyms. To cover a broader range of studies and mitigate the search bias, we prioritize systematic literature reviews on the search. For instance, we search for software bug relationships using the search string "systematic literature review software bugs".

In addition, we search the literature for associations related to the sub-practices of continuous integration — tests practices, integration frequency, build health maintenance, and quick fixes of broken builds [12], [26], [39]. In the full search, we selected 39 studies, of which 12 are systematic literature reviews. All the selected studies are referenced, and a complete list is available in our replication package[2].

Continuous Integration and Software Quality: Soares et al. [15] conduct a systematic literature review on the associations between CI and software development as a whole. The review highlights associations between CI and an increase in bug/issue resolution [5]. For this reason, our literature-based DAG starts from the association between continuous integration and bug resolution. The notation $CI \rightarrow BugResolution$ represents a causal flow from CI to Bug Resolution. We expand our DAG based on other studies that raise other diverse CI associations (as summarized in Table I).

CI is related to a team's capability of discovering more bugs than no CI teams, and CI projects tend to present fewer defects ($CI \rightarrow BugReport$) [8], [44], as well as CI is associated with an increase in transparency ($CI \rightarrow Transparency$) [38]. Another study points out that CI induces a decrease in the number of messages into pull requests ($CI \rightarrow Communication$) [37]. CI is also associated with a false sense of confidence ($CI \rightarrow Overconfidence$) [19], [20] and technical challenges related to environment configurations ($CI \rightarrow TechnicalChallenges$) [2], [19]. Table II summarizes these associations from CI to co-variables.

Confounders related to bug reports: To investigate potential confounders to the effect of $CI \rightarrow BugReport$, we search for other factors associated with $BugRerport$. Table III shows the associations extracted from a taxonomy [17], a systematic literature review [1], and a mixed mining software repositories (MSR)-survey study [44].

Confounders related to bug resolution: To investigate potential confounders to the effect of $CI \rightarrow BugResolution$, we search for other factors associated with $BugResolution$ in the literature. We found that $Maintainability$, $Analysability$, $Changeability$,

[1]https://scholar.google.com/

[2]https://anonymous.4open.science/status/ci_quality_study_replication

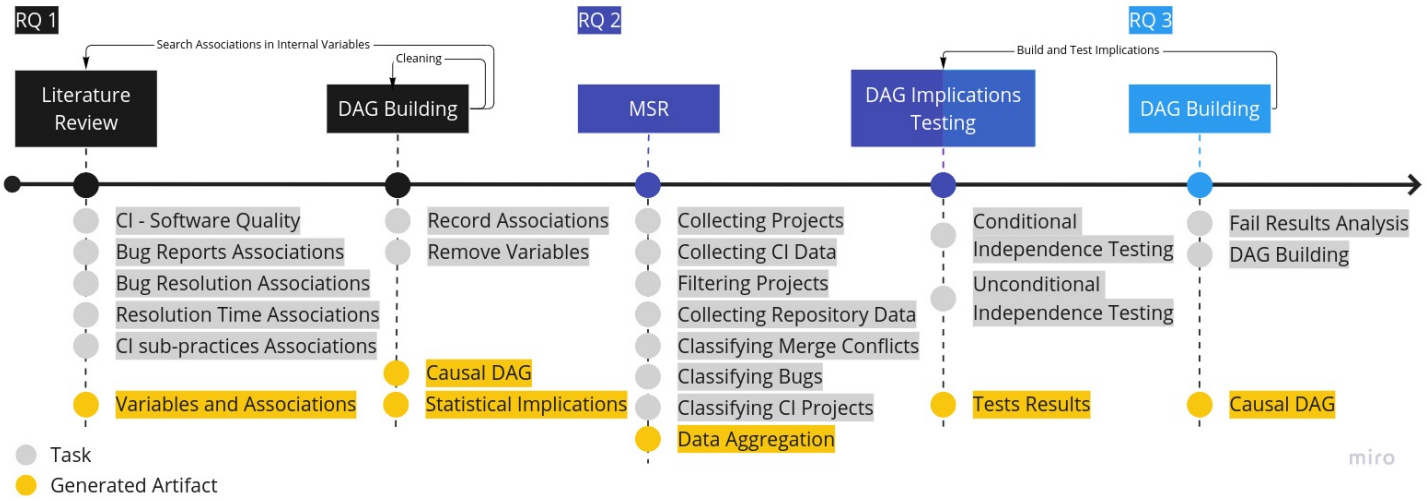

Fig. 1. Research method pipeline.

$Stability$, $Testability$, $ProjectVolume$, $Duplication$, $Unitsize$, $Unitcomplexity$, and $Modulecoupling$ all share an association with $BugResolution$. For the sake of readability, we group all these relationships into $InternalQuality \rightarrow BugResolution$ [13]. We also found in the literature review from Zhang et al. [43] the associations $Communication \rightarrow BugResolution$ and $IssuePriority \rightarrow BugResolution$. Table IV shows all these associations and their rationales.

Confounders related to resolution time: Table V shows associations between other factors than CI and $ResolutionTime$. We found associations extracted from the literature including variables $IssueType$, $Communication$ (e.g., comments in issues, bug reports, pull requests), $IssuePriority$, $CommitSize$, $OperateSystem$, and $IssueDescription$.

Internal confounders: We also consider internal relationships between every discovered variable to understand possible confounding scenarios in our causal DAG. That means considering potential associations between peripheral variables, e.g., $IssueType \rightarrow CommitSize$. Table VI shows such potential associations and their rationales. With these previous associations, we build the partial causal DAG represented in Fig. 2, where continuous integration is the intervention, and $BugResolution$ and $BugReport$ are potential outcomes.

Confounders related to Continuous Integration and its sub-practices: To investigate variables related to continuous integration (CI), we also consider its sub-practices [12], [26], [39]. We consider the practices: tests practices, integration frequency, build health maintenance, and time to fix a broken build. We represent an association with a sub-practice of CI as an association with CI itself. For example, $AutomatedTests \rightarrow Confidence$ will be represented as $ContinuousIntegration \rightarrow Confidence$. This decision avoids intangible discussions about what comes first, CI or Automated Tests, for instance. To implement CI, we consider

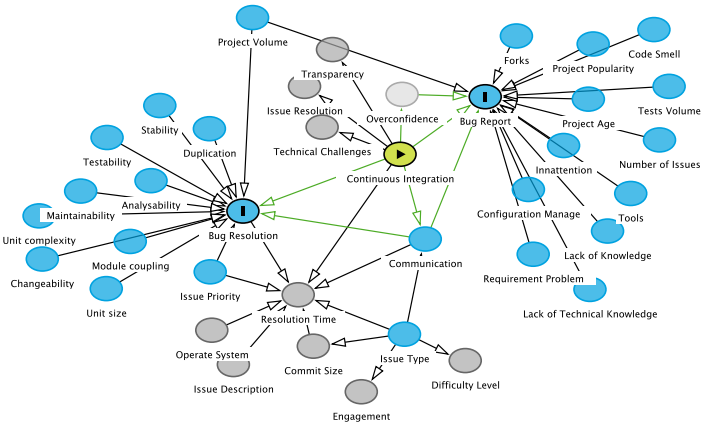

Fig. 2. Partial causal DAG for bug reports associations with interest covariables.

that such practices are implicit, i.e., there is no CI employment without implementing those practices [15], [39]. Therefore, distinguishing the effect of CI and the effect of its sub-practices would be challenging. Table VII groups and presents the associations relating to testing and build practices.

With associations shown in Table VII we build the partial causal DAG represented in Fig. 3, where $AutomatedTests$ is the intervention. The relationship $AutomatedTests \rightarrow BugReport$ was omitted in this figure because it is a partial DAG centered in the $AutomatedTests$. While with Table VIII we build the partial causal DAG represented in Fig. 4, centered in the build attributes.

Fig. 5 shows the unified and complete literature-based causal DAG, i.e., the union of the assumptions cataloged in Table I. The literature-based causal DAG expresses all known associations for the related variables in the studied domain. The nodes on the DAG represent variables (e.g., Continuous Integration and Bug Report), and the directed edges connecting the variables represent an association between them. The

TABLE I
ASSOCIATIONS IDENTIFIED IN THE LITERATURE ABOUT CI, SOFTWARE QUALITY AND ADJACENT RELATIONSHIPS.

CI and software quality associations	
Association	**References**
$CI \rightarrow BugResolution*$	[5]
$CI \rightarrow ResolutionTime$	[23], [47]
$CI \rightarrow BugReport$	[8], [44]
$CI \rightarrow Transparency$	[38]
$CI \rightarrow Communication$	[37]
$CI \rightarrow Overconfidence$	[19], [20]
$CI \rightarrow TechChallenges$	[2], [19]
Associations related to bug reports	
$TestsVolume \rightarrow BugReport$	[44]
$LackOfKnowledge \rightarrow BugReport$	[17]
$CodeSmells \rightarrow BugReport$	[1]
$LackTechKnowledge \rightarrow BugReport$	[17]
$RequiremProblem \rightarrow BugReport$	[17]
$Overconfidence \rightarrow BugReport$	[17]
$Inattention \rightarrow BugReport$	[17]
$Communication \rightarrow BugReport$	[17]
$ConfigManagement \rightarrow BugReport$	[17]
$Tools \rightarrow BugReport$	[17]
$NumberOfForks \rightarrow BugReport$	[44]
$QuantIssues \rightarrow BugReport$	[44]
$ProjPopularity \rightarrow BugReport$	[44]
$ProjAge \rightarrow BugReport$	[44]
Associations related to automated build	
$TeamSize \rightarrow BuildHealth$	[38]
$MultipleWorkspace \rightarrow BuildHealth$	[38]
$DeveloperRole \rightarrow BuildHealth$	[38]
$CommitSize \rightarrow BuildHealth$	[34], [38], [42].
$CommitType \rightarrow BuildHealth$	[34], [38], [42].
$CommitMoment \rightarrow BuildHealth$	[38].
$TeamDistribution \rightarrow BuildHealth$	[38].
$Tools \rightarrow BuildHealth$	[22].
$ExtraComplexity \rightarrow BuildHealth$	[16].
$FlakyTests \rightarrow BuildHealth$	[16], [42].
$ContributorType \rightarrow BuildHealth$	[42].
$TimeToFix \rightarrow Costs$	[38].
$Communication \rightarrow TimeToFix$	[38].
$DeveloperRole \rightarrow TimeToFix$	[38].
$CommitType \rightarrow TimeToFix$	[38].
$IntegrationFreq \rightarrow TimeToFix$	[38].
$ProgramLanguage \rightarrow TimeToFix$	[22].
$ErrorUnderstand \rightarrow TimeToFix$	[10].
$BuildFailType \rightarrow ErrorUnderstand$	[10]
$BuildHealth \rightarrow WorkTime$	[38]
$BuildHealth \rightarrow MergeConflicts$	[16]
$FixTools \rightarrow ErrorUnderstand$	[10]
$BuildFailType \rightarrow TimeToFix$	[10]
$TestsVolume \rightarrow CommitSize$	[16]
$ContributorType \rightarrow CommitSize$	[35]
$ContributorType \rightarrow AutomatedTests$	[35]
$Extracomplexity \rightarrow ContributorType$	[35]
$CommitSize \rightarrow MergeConflicts$	[16].
Associations related to resolution time	
$IssueType \rightarrow ResolutionTime$	[4], [18], [33], [41]
$Communication \rightarrow ResolutionTime$	[18], [27]
$IssuePriority \rightarrow ResolutionTime$	[18]
$CommitSize \rightarrow ResolutionTime$	[18]
$OperateSystem \rightarrow ResolutionTime$	[18]
$IssueDescription \rightarrow ResolutionTime$	[18]
Internal associations	
$IssueType \rightarrow Engagement$	[41]
$IssueType \rightarrow InfoSharing$	[41]
$IssueType \rightarrow Communication$	[41]
$IssueType \rightarrow DifficultyLevel$	[33]
$Stability \rightarrow TechChallenges$	[2]
$IssueType \rightarrow CommitSize$	[3]
Associations related to bug resolution	
$InternalQuality \rightarrow BugResolution*$	[17]
$Communication \rightarrow BugResolution*$	[43]
$IssuePriority \rightarrow BugResolution*$	[43]
Associations related to test practices	
$AutomatedTests \rightarrow BugReport$	[14].
$AutomatedTests \rightarrow CodeCoverage$	[14].
$AutomatedTests \rightarrow WorkTime$	[14].
$AutomatedTests \rightarrow Confidence$	[14].
$AutomatedTests \rightarrow HumanEffort$	[14].
$AutomatedTests \rightarrow Cost$	[14].
$AutomatedTests \rightarrow BugDetection$	[14].
$AutomatedTests \rightarrow TechChallenges$	[45].
$LackTechKnowledge \rightarrow AutomatedTests$	[45].
$Stability \rightarrow TechChallenges$	[45].
$Design \rightarrow TestReusability$	[14].
$TestRepetition \rightarrow Reliability$	[14].
$ProjAge \rightarrow AutomatedTests$	[6], [7], [21], [32].

* In the final version of the DAG, we combine $BugResolution$ into $BugReport$.

associations "flow" from one variable to another. For instance, according to Fig. 5, Continuous Integration influences Bug Report. Note that the variables $AutomatedTests$, $BuildHealth$, $IntegrationFrequency$, and $TimeToFix$ are all represented by $ContinuousIntegration$. This causal DAG built upon the existing literature is the starting point for identifying variables that must be measured and controlled to allow a causal analysis of the influence of CI on software quality.

*2) **DAG Building** :* We built the literature-based DAG in Fig. 5 expressing the associations from the Table I. To analyze the causal effect of CI on software quality, we need to be attentive to confounding effects, i.e., bias due to backdoor paths. Then, we need to draw a DAG containing a sufficient set of variables that show the existing backdoor paths.

Based on Reichenbach's Common Cause Principle and the Markov Condition, we know that a causal DAG should include the common causes of any pair of variables in the DAG [24], [28]. Therefore, we can discard several variables in Fig. 5 because they are external variables without connecting with the variables under investigation (i.e., CI and $BugReport$), as well they are not common causes for any variable selected for the analysis (i.e., CI, Bug Report or one of its common

TABLE II
ASSOCIATIONS IDENTIFIED IN THE LITERATURE ABOUT CI AND SOFTWARE QUALITY VARIABLES.

Association	Rationale
$CI \rightarrow BugResolution$	Projects presents more resolved issues and bugs after adoption of CI [5].
$CI \rightarrow ResolutionTime$	CI is related to an increasing in the number of issues closed by period, helping to spend less time debugging and more time adding features [23], [47].
$CI \rightarrow BugReport$	CI teams discover more bugs than no-CI teams, and CI projects present fewer defects than no-CI projects [8], [44].
$CI \rightarrow Transparency$	CI is associated with a transparency increase, facilitating collaboration [38].
$CI \rightarrow Communication$	The general discussion, the number of line-level review comments, and change-inducing review comments tend to decrease after CI adoption without affecting pull request activity [37].
$CI \rightarrow Overconfidence$	CI developers are reported as suffering from a false sense of confidence (when blindly trusting the tests) [19], [20].
$CI \rightarrow TechnicalChallenges$	Configuring the build environment, the tools, and practices impose challenges for CI teams [2], [19].

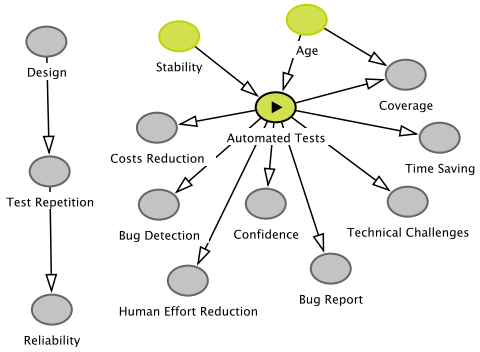

Fig. 3. Partial causal DAG for automated tests associations with interest co-variables.

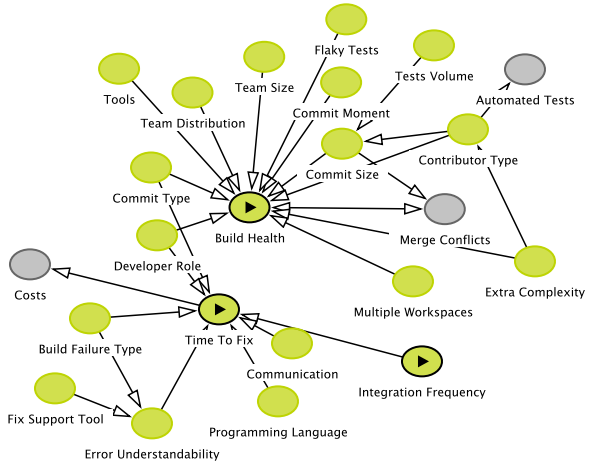

Fig. 4. Partial causal DAG for build attributes and their associations with interest co-variables.

TABLE III
ASSOCIATIONS IDENTIFIED IN THE LITERATURE ABOUT BUG REPORTS.

Association	Rationale
$LackOfKnowledge \rightarrow BugReport$	Insufficient domain and linguistic knowledge are presented as possible human root causes for software defects [17].
$LackTechKnowledge \rightarrow BugReport$	Insufficient programming and strategy knowledge and failure to catch the specific feature of the problems are mapped as possible human root causes for software defects [17].
$RequiremProblem \rightarrow BugReport$	Requirement management problems and a misunderstanding of requirements and design specifications are reported as possible human causes of software defects [17].
$Overconfidence \rightarrow BugReport$	Overconfidence and confirmation bias contributes to evaluation errors and software defects [17].
$Inattention \rightarrow BugReport$	Interruptions and other kinds of inattention are reported as possible human causes of software defects [17].
$Communication \rightarrow BugReport$	Communication problems lead to expression and comprehension errors [17].
$ConfigManagement \rightarrow BugReport$	Configuration management problems lead to process errors [17].
$Tools \rightarrow BugReport$	Tools problems like compiler induced defects are possible root causes of software defects [17].
$CodeSmells \rightarrow BugReport$	Code smells on the occcurrence of bugs [1].
$NumberOfForks \rightarrow BugReport$	The number of forks has an association with an increase in bug reports [44].
$ProjAge \rightarrow BugReport$	Project age has a significant negative effect on the count of bugs reported by core developers [44].
$ProjPopularity \rightarrow BugReport$	Project's popularity has a significant negative effect on the count of bugs reported by core developers [44].
$QuantIssues \rightarrow BugReport$	The number of non-bug issue reports has a significant and positive effect on the response [44].
$TestsVolume \rightarrow BugReport$	The size of test files has a negative effect on bug reports [44].

TABLE IV
ASSOCIATIONS IDENTIFIED IN THE LITERATURE ABOUT BUG RESOLUTION.

Association	Rationale
$InternalQuality \rightarrow BugResolution$	Elements of $InternalQuality$, such as maintainability, analysability, changeability, stability, testability, project volume, duplication, unit size, unit complexity, and module coupling, present significan correlation with defect resolution efficiency [17].
$Communication \rightarrow BugResolution$	Human and data elements such as comments, severity, product, component, among others, can improve the performance of bug resolution [43].
$IssuePriority \rightarrow BugResolution$	Priority and severity are non-textual factors of a bug report that enhance the capability of bug resolution [43].

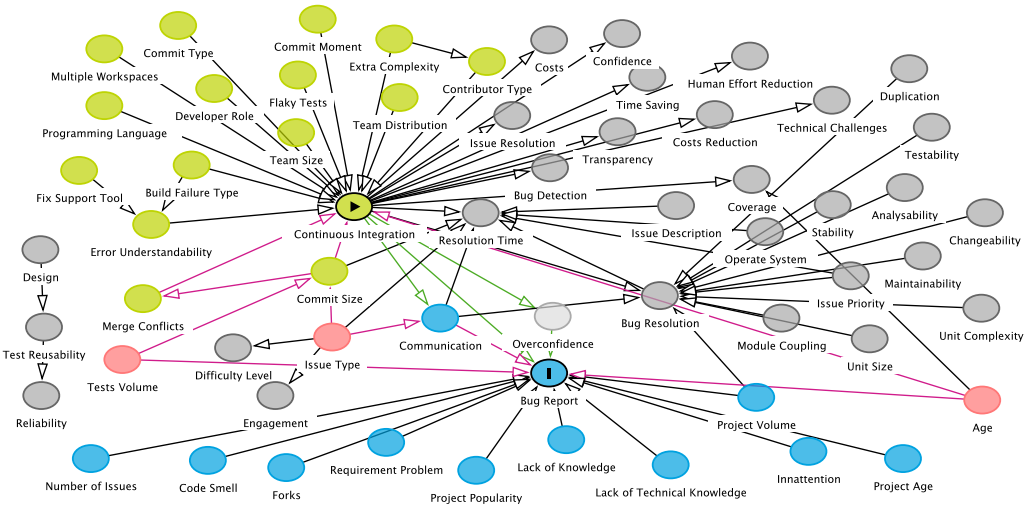

Fig. 5. Complete theory-driven causal DAG for CI, Bug Reports and their co-variables.

TABLE V
ASSOCIATIONS IDENTIFIED IN THE LITERATURE ABOUT THE RESOLUTION TIME.

Association	Rationale
$IssueType \rightarrow ResolutionTime$	Issue fixing times are different for different issue types [4], [18], [33], [41].
$Communication \rightarrow ResolutionTime$	The number of comments and the max length of all comments in the bug reports impact the resolution time. Bugs with little discussion tend to be resolved quickly [18], [27].
$IssuePriority \rightarrow ResolutionTime$	The severity of a bug report influences the delay before fixing it. As high the severity level, the fewer the delay [18].
$CommitSize \rightarrow ResolutionTime$	The size of code churn (number of methods) impacts the delay before fixing a bug report [18].
$OperateSystem \rightarrow ResolutionTime$	The median delay before fixing a bug found on Linux is shorter than other OS [18].
$IssueDescription \rightarrow ResolutionTime$	Increasing the literal length of the bug report description can increase delay until the team checks it as resolved [18].

TABLE VI
INTERNAL ASSOCIATIONS CATALOGED AMONG THE LITERATURE REGARDING THE DISCOVERED VARIABLES.

Association	Rationale
$IssueType \rightarrow CommitSize$	The issue type is associated with the size of the code churn [3].
$IssueType \rightarrow Engagement$	Developers tend to spend more effort engaging with one another regarding new features and software extensions than in defects [41].
$IssueType \rightarrow InfoSharing$	Developers tend to share more information on defects and enhancements than support tasks [41].
$IssueType \rightarrow Communication$	A higher number of comments is associated with enhancements and defects [41].
$IssueType \rightarrow DifficultyLevel$	There is an association between the difficulty of a change and its type [33].
$Stability \rightarrow TechnicalChallenges$	The maturity of the tools, infrastructure, and CI activities imposes challenges to practitioners [2].

TABLE VII
CI ASSOCIATIONS CATALOGED AMONG THE LITERATURE FROM THE PERSPECTIVE OF TEST PRACTICES.

Association	Rationale
$AutomatedTests \rightarrow BugReport$	Automated tests are related to improved product quality in terms of fewer defects in the software [14].
$AutomatedTests \rightarrow CodeCoverage$	Automated tests are related to high coverage of code [14].
$AutomatedTests \rightarrow WorkTime$	Automated tests are related to reduced testing time [14].
$AutomatedTests \rightarrow Confidence$	Automated tests are related to increased confidence in the quality of the system [14].
$AutomatedTests \rightarrow HumanEffort$	Automated tests are related to the less human effort that can be redirected for other activities [14].
$AutomatedTests \rightarrow Cost$	Automated tests are related to a reduction in cost [14].
$AutomatedTests \rightarrow BugDetection$	Automated tests are related to increased fault detection [14].
$AutomatedTests \rightarrow TechnicalChallenges$	Automated tests require different skills to implement them effectively [45].
$LackTechKnowledge \rightarrow AutomatedTests$	The skills level of testers could be a hindrance to test automation [45].
$Stability \rightarrow TechnicalChallenges$	The stability and maturity of the software under test affect the maintenance effort of tests [45].
$Design \rightarrow TestReusability$	Designing tests with maintenance in mind, they can be repeated frequently [14].
$TestRepetition \rightarrow Reliability$	When repeating tests, they are more reliable than single executions [14].
$ProjAge \rightarrow AutomatedTests$	The number, coverage, and maturity of automated tests increase with time [6], [7], [21], [32].

causes). For instance, $ProjectPopularity$ is associated with $BugReport$ but does not have an association with no other variable on the DAG. Then $ProjectPopularity$ can be discarded from our causal analysis. On the other hand, Age is a common cause associated with $ContinuousIntegration$ and $BugReport$, and it is essential to analyze the causal effect between them since Age is potentially a confounding factor.

TABLE VIII
CI ASSOCIATIONS CATALOGED AMONG THE LITERATURE FROM THE PERSPECTIVE OF BUILD PRACTICES.

Association	Rationale
$BuildHealth \rightarrow WorkTime$	Broken builds lead to loss of time by freezing development and tests [38].
$BuildHealth \rightarrow MergeConflicts$	Broken builds lead to work blockage, which in turn leads to merge conflicts [16].
$TeamSize \rightarrow BuildHealth$	The team size relates to build breakage. Shorter teams tend to break fewer than larger ones [38].
$MultipleWorkspace \rightarrow BuildHealth$	Maintaining multiple physical structures for multiple branches is associated with more build breakage [38].
$DeveloperRole \rightarrow BuildHealth$	There is a statistical difference in build breakage among different role groups [38].
$CommitSize \rightarrow BuildHealth$	The size of the changes is related to a higher probability of build failure [34], [38], [42].
$CommitType \rightarrow BuildHealth$	The commit type (such as features and bugs) and the contribution model (e,g., pull request and push model) are associated with build breakage [34], [38], [42].
$CommitMoment \rightarrow BuildHealth$	There is an association between the moment of contributions and the rate of build breakage [38].
$TeamDistribution \rightarrow BuildHealth$	The geographical distance of the team members is associated with the build results [38].
$Tools \rightarrow BuildHealth$	The languages and their tools are related to different build breakage rates [22].
$ExtraComplexity \rightarrow BuildHealth$	Complex builds tend to break [16].
$FlakyTests \rightarrow BuildHealth$	Flaky tests favor the occurrence of build breakage [16], [42].
$ContributorType \rightarrow BuildHealth$	Less frequent contributors tend to break builds less [42].
$TimeToFix \rightarrow Costs$	The time lost relates directly to a monetary cost [38].
$Communication \rightarrow TimeToFix$	The feedback mechanisms and information speed affect the awareness of a broken build and the time to fix it [38].
$DeveloperRole \rightarrow TimeToFix$	The developer role is associated with the time to fix a broken build [38].
$CommitType \rightarrow TimeToFix$	The characteristics of the branches and code access (e.g., isolated branches) are associated with the time to fix a broken build [38].
$IntegrationFreq \rightarrow TimeToFix$	The integration frequency in the team affects the build fixing [38].
$ProgramLanguage \rightarrow TimeToFix$	The programming language is related to the time spent to fix a broken build [22].
$ErrorUnderstand \rightarrow TimeToFix$	The understandability of the build failures directly impacts the time needed to solve them [10].
$BuildFailType \rightarrow TimeToFix$	The build failure types are associated with different difficulty levels [10].
$TestsVolume \rightarrow CommitSize$	Complex and time-consuming testing is a possible reason for large commits [16].
$ContributorType \rightarrow CommitSize$	The type of contributor (e.g., casual) relates to the build breakage rate [35].
$ContributorType \rightarrow AutomatedTests$	The contributor type is related to the number of automated tests [35].
$Extracomplexity \rightarrow ContributorType$	The complexity of the jobs is related to the type of contributor in the projects [35].
$CommitSize \rightarrow MergeConflicts$	Large commits are associated with merge conflicts [16].
$FixTools \rightarrow ErrorUnderstand$	Fix support tools improves the understandability of the build logs [10].
$BuildFailType \rightarrow ErrorUnderstand$	The build failure type is associated with different levels of understandability [10].

*B. **RQ2. Is the causal effect of CI on software quality empirically observable?***

Based on the concepts of conditional independence and the d-separation rules, it emerges a set of statistically testable restrictions as implications of a model [24]. Such restrictions (i.e., statistical implications) are conditional or unconditional independencies between DAG variables that must be found in any dataset generated by the causal processes described in the DAG. By building a dataset with the variables in our causal DAG, we can test the implications of the DAG statistically.

The implications are in the form of unconditional independencies, like $Age \perp\!\!\!\perp Commit_{S}ize$, which means Age is independent of $CommitSize$. Alternatively, the implications may have the form of conditional independence, like $MergeConflicts \perp\!\!\!\perp TestsVolume \mid CommitSize$, which means $MergeConflicts$ is independent of $TestsVolume$ conditioned in $CommitSize$. Since we use the DAGitty R package [25] to draw our causal DAGs, we obtain from the *impliedConditionalIndependencies* function a list of testable implications that become our causal hypotheses.

We then perform a mining software repository study to collect data on the selected variables and allow us to test the causal DAG statistical implications, i.e., our causal hypotheses.

*1) **Mining Software Repository**:* Based on the knowledge and assumptions acquired in the literature review and the causal DAG built to answer the RQ1, we have support for building a dataset to analyze the relationship between CI and software quality, including confounding variables. Using the dataset, we can check the validity of our DAG through statistical tests.

Selecting Projects. We obtained the initial project list searching for the most starred repositories in the GitHub Search API.[3] The search included 15 popular languages: C, C#, C++, Go, Java, JavaScript, Kotlin, Objective-C, PHP, Python, Ruby, Rust, Scala, Swift, and TypeScript. At the end of this search, we had a list of 11,671 project repositories.

We also search projects data from SonarQube web API[4] considering the 15 mentioned languages, ordering by the number of lines of code. After that, we select projects hosted on GitHub, obtaining 16,214 more projects, summing up 27,885 project repositories.

Collecting Data on Continuous Integration Online Service. Inspired by Hilton's work [31], which searches for the usage of 5 different services — Travis CI, CircleCI, AppVeyor, Werker, and Cloud-Bees/ Jenkins CI, we search for the same services, but adding a new one — GitHub Actions.[5]

We discovered a high CI service usage (54.3%) among the collected projects, being Travis CI[6] the most popular service. Table IX shows the number of projects found for each CI Service as well as the criteria of identification applied. For projects using Travis CI, we also record the date when the

[3]https://docs.github.com/en/rest/reference/

[4]https://community.sonarsource.com/t/list-of-all-public-projects-on-sonarcloud-using-api/33551

[5]https://github.com/features/actions

[6]https://travis-ci.org/

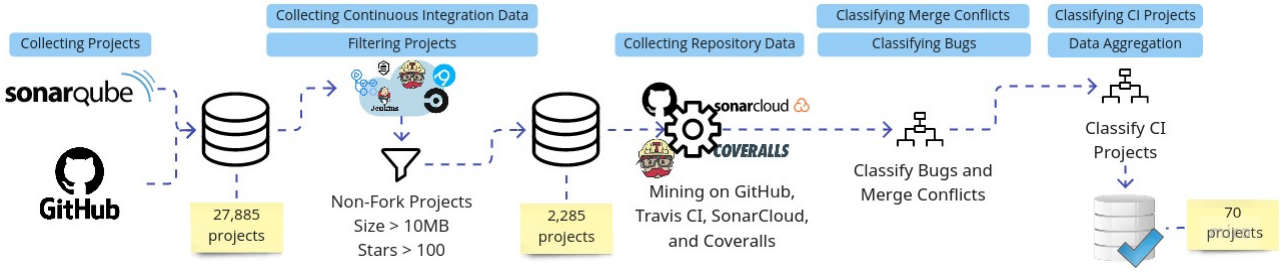

Fig. 6. Mining Software Repository

TABLE IX
THE CI SERVICE USAGE ON THE DATASET AND THE CLASSIFICATION CRITERIA.

CI Service	Qty	Criteria
Travis CI	9,092 (32.6%)	We search on Travis API[a] for the existence of the project in the service. If true, we search on GitHub for the existence of a *".travis.yml"* file.
GitHub Actions	5,630 (20.1%)	Using the search code feature of GitHub API, we search for the extension *".yml"* in the path *".github/workflows"*.
Circle CI	307 (1.1%)	Through Circle CI API[b] we search for the existence of the project on the service. If not located, we searched for the file *"config.yml"* in the path *".circleci"*.
Jenkins	58 (0.2%)	Using the search code feature of GitHub API, we search for the file *"Jenkinsfile"*.
AppVeyor	29 (0.1%)	Using the search code feature of GitHub API, we search for the file *"appveyor.yml"*.
Wercker	0	Using the Wercker API[c] we search for the existence of the project on the service.
No CI Service	12,769 (45.7%)	

[a]https://docs.travis-ci.com/user/developer/#api-v3
[b]https://circleci.com/docs/api/v2/
[c]https://devcenter.wercker.com/development/api/endpoints/

service was firstly configured, i.e., the date of the first commit of the file *".travis.yml"*.

Afterwards, we filtered out irrelevant repositories. In particular, consider only non-fork projects with more than 100 stars. To avoid toy projects, we filter projects with a size greater than 10MB [29], [30]. Finally, we consider only projects either using Travis CI or projects not using CI at all. We chose Travis CI as it is the most used service in our dataset. In addition, Travis CI has a public API to obtain software builds data. After applying the filtering criteria, our sample reduced from 27,885 to 2,285 repositories, i.e., 1,515 repositories using Travis CI and 770 without CI.

Collecting Data on Pull Requests and Issues. Using the endpoint "pulls" and "issues" from the GitHub API, we collected all pull requests and issues in the repositories. This process resulted in a total of 1,425,493 pull requests and 1,050,371 issues. Finally, we select projects with 12 consecutive months of activity, i.e., having issues and pull requests in all months. We obtain 537 repositories (i.e., 498 using Travis CI and 39 without a CI service).

Collecting Comments of Pull Requests. We consider only pull requests in the 12 months period of activity. Using the GitHub API, we collect all simple and review comments, as well as associated information regarding authors and pull requests. All these data allow us to compute a metric to measure $Communication$ and $MergeConflicts$.

Collecting Commits of Pull Requests. We collect commit sha, date, message, and author for each considered pull request. To collect the detailed commit information regarding lines of code added, removed, or changed, we applied a two-step strategy: 1) We adapted RepoDriller[7] to mine the size of a commit, the number of lines of code added, removed, the number of files modified, and the test files and test lines; 2) for commits that we could not count the changes, we developed a web scraping script to mine the changes from the commit pages of the repositories. These two steps are complementary, aiming to collect data for all commits. The commits data help us to compute metrics to measure $CommitSize$, $TestsVolume$, and $MergeConflicts$.

Classifying Merge Conflicts. To identify the merge conflicts occurrences, we consider the data from commits, merge-commits, and the messages from pull requests. We apply heuristics in the log or text of these data. We consider two alternative patterns of messages containing:

- ("MERGE" OR "MERGING") AND "CONFLICT"
- "FIX MERGE" OR "MERGE FIX"

We find a total of 7,423 merge conflicts in 1,425,493 pull requests, which corresponds to 0.5% of occurrences.

Classifying Bugs. To classify issues as bugs, we consider only projects using GitHub issues. We mapped the labels used in their repositories through a careful semi-manual inspection (a preliminary parse was performed by a script available in our online replication package[8]).

The exception to our primary approach is a set of 10 projects having a high number of issues without any labels. To avoid missing these data, we adopt a conservative approach classifying as bugs issues containing the terms "bug" or "fix" in their title or their body. We classified 90,159 out of 1,050,371 issues as bugs, representing 8.5% of occurrences.

Collecting Data on Software Builds. Using the Travis API, we collect information about the builds of the projects, e.g., the status and duration. We collect data on 211,281 software builds for 451 repositories.

[7]https://github.com/mauricioaniche/repodriller
[8]https://anonymous.4open.science/status/ci_quality_study_replication

Collecting Data on Code Coverage. To collect the code coverage of the builds, we search for coverage information in two different services — Coveralls[9], and SonarCloud[10]. We found 81 projects using Coveralls, 17 using Sonar Cloud, and 4 with data in both services.

Classifying CI Projects. In order to avoid analyzing CI Theater [11], [46] and also observing recommendations from Soares et al. [15], in addition to the use of Travis CI, we consider build and code coverage information, meaning that projects have actual build and test activity.

Selecting projects having coverage and build data, we found 35 projects. These projects have a median of 81.05% of code coverage. Regarding build activity (i.e., frequency of days having builds in a given period) the projects have a median of 5.9%. With respect to build health (i.e., rate of successful builds) the projects have an average of 21% of successful builds. Lastly, these projects have a median time to fix a broken build of 350,763 seconds (i.e., 4 days, which is reasonable for open source projects). After achieving a set of 35 CI projects obeying all filtering stages, we draw 35 out of 39 non-CI projects to balance our dataset, ending with 70 projects. After achieving a set of 35 CI projects obeying all filtering stages, we draw 35 out of 39 non-CI projects to balance our dataset. Table X shows a summary of the final dataset.

Data Aggregation. With the collected data, we can analyze the Commit Size, Test Volume, Merge Conflicts, Communication, Bug Reports, and Age variables. We can not measure Issue Type and Overconfidence for two reasons. First, classifying $IssueType$ from the collected issue data is challenging given projects do not have any consistent pattern (e.g., standardized tags, such as "enhancement", "perfective", "corrective"). Second, overconfidence is a subjective feeling not feasible to measure through mining software repositories. Therefore, we consider these two variables as *latent* (i.e., unmeasured). Latent variables can still be represented and analyzed in a causal DAG even though we cannot statistically test latent variables. However, we can still interpret them [28].

We compute the metrics aggregating into chunks representing development months. Aggregating enables us to count our metrics. The decision by monthly chunks is due to months to be more consistent than another time period (e.g., week), especially for open-source projects, which have fewer activity levels. All the 70 projects have a period of 12 months with data mined. For each month, we compute the following metrics:

- **Commit Size** is the average of the size of the commits in a month. We compute the commit size as the sum of lines of code added and removed in each contribution.
- **Communication** is the sum of comments and review comments in the pull requests. We consider the average of communication in a month.
- **Merge Conflicts** is the number of merge conflicts in a month.

[9]https://coveralls.io/

[10]https://sonarcloud.io/

TABLE X
SUMMARY OF THE DATA SET.

Number of Projects	70
Number of Pull Requests	41,856
Number of Commits	29,127
Number of Builds	9,194
Number of Bugs	1,173
Number of Merge Conflicts	189

- **Age** is the number of days of a repository from its creation until the first day of each month.
- **Test Volume** is the proportion of lines of test code modified (added, removed, or changed) in the commits. We consider the median test volume in a month.
- **Bug Reports** is the number of bugs reported in a month.
- **CI** is a binary categorical variable indicating whether the project uses CI or not.

*2) **DAG Implications Testing**:* Having the dataset and the set of testable causal hypotheses, we can investigate if the implications from the causal DAG are actually held in the empirical dataset. We test the unconditional independencies causal hypotheses (e.g., $X \perp\!\!\!\perp Y$) using the *dcov.test* function from the energy R package. [11] This test verifies if two variables on the dataset are independent. We test the conditional independence causal hypotheses (e.g., $X \perp\!\!\!\perp Y \mid Z$) with the *Kernel conditional independence test* from CondIndTests R package [9]. Both tests are non-parametric and suitable for our data.

C. RQ3. What would be an accurate causal theory for CI?

Considering the investigations in RQ1 and RQ2, we have information regarding which independency tests have passed or failed in our analyses. This gives us the opportunity to propose adaptations to the causal DAG built in RQ1. For example, by testing the data, we can observe a dependence between Age and $Communication$ variables, then we propose a new edge $Age \rightarrow Communication$. We build a new causal DAG by combining the knowledge from the literature (RQ1) and the statistical tests on the dataset (RQ2). We call this hybrid approach as theory-data causal DAG. The theory-data causal DAG is a DAG that is compatible with the dataset in terms of the independency relationships between the variables.

There are dozens of algorithms for causal discovery from data, such as PC [40] that rely on a set of conditional independence tests to discover the causal structure from the data. On the other hand, Cartwright [36] argues that causal investigation requires background information too (i.e., causal assumptions)—"No causes in, no causes out".

Thus, to implement the theory-data causal DAG, we start from the original causal DAG built in RQ1 (i.e., causal assumptions from literature) and the knowledge acquired in RQ2 (i.e., the independency relationships that were not present in the data). Next, we structure our methodology procedures in three steps.

[11]https://cran.r-project.org/package=energy

Step 1. By analyzing the rejected hypotheses in RQ2, in conjunction with d-separation rules, we can infer associations that should be added or removed in the theory-data causal DAG.

Step 2. We generate the theory-data causal DAG after identifying the disconnected vertices and the necessary changes from Step 1.

Step 3. We test the d-separation implications on the theory-data causal DAG. If any test fails, we go back to Step 1 and refine the DAG.

REFERENCES

[1] A. Cairo, G. Carneiro, and M. Monteiro, "The impact of code smells on software bugs: A systematic literature review", Information, vol. 9, no. 11, p. 273, 2018.

[2] A. Debbiche, M. Dienér, and R. Berntsson Svensson, "Challenges when adopting continuous integration: A case study," Product-Focused Software Process Improvement, pp. 17–32, 2014.

[3] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?," Proceedings of the 2008 international workshop on Mining software repositories - MSR '08, 2008.

[4] A. Murgia, G. Concas, R. Tonelli, M. Ortu, S. Demeyer, and M. Marchesi, "On the influence of maintenance activity types on the issue resolution time," Proceedings of the 10th International Conference on Predictive Models in Software Engineering, 2014.

[5] A. Rahman, A. Agrawal, R. Krishna, and A. Sobran, "Characterizing the influence of continuous integration: Empirical results from 250+ open source and proprietary projects," Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics, 2018.

[6] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of Production & Test code," 2008 1st International Conference on Software Testing, Verification, and Validation, 2008.

[7] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through Repository Mining," Empirical Software Engineering, vol. 16, no. 3, pp. 325–364, 2010.

[8] C. Amrit and Y. Meijberg, "Effectiveness of test-driven development and continuous integration: A case study," IT Professional, vol. 20, no. 1, pp. 27–35, 2018.

[9] C. Heinze-Deml, J. Peters, and N. Meinshausen, "Invariant causal prediction for nonlinear models," arXiv.org, 19-Sep-2018. [Online]. Available: https://arxiv.org/abs/1706.08576v2. [Accessed: 06-Apr-2022].

[10] C. Vassallo, S. Proksch, T. Zemp, and H. C. Gall, "Every build you break: Developer-oriented assistance for build failure resolution," Empirical Software Engineering, vol. 25, no. 3, pp. 2218–2257, 2019.

[11] "CI Theatre: Technology Radar," Thoughtworks. [Online]. Available: https://www.thoughtworks.com/radar/techniques/ci-theatre. [Accessed: 22-Mar-2022].

[12] "Continuous integration," martinfowler.com. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html. [Accessed: 22-Mar-2022].

[13] D. Bijlsma, M. A. Ferreira, B. Luijten, and J. Visser, "Faster issue resolution with higher technical quality of software," Software Quality Journal, vol. 20, no. 2, pp. 265–285, 2011.

[14] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, K. Petersen, and M. V. Mantyla, "Benefits and limitations of Automated Software Testing: Systematic Literature Review and practitioner survey," 2012 7th International Workshop on Automation of Software Test (AST), 2012.

[15] E. Soares, G. Sizilio, J. Santos, D. Alencar da Costa, U. Kulesza, "The effects of continuous integration on software development: a systematic literature review", Empir Software Eng 27, 78 (2022). https://doi.org/10.1007/s10664-021-10114-1

[16] E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery—a systematic literature review," Information and Software Technology, vol. 82, pp. 55–79, 2017.

[17] F. Huang, B. Liu, and B. Huang, "A taxonomy system to identify human error causes for software defects". In The 18th international conference on reliability and quality in design, p. 44, 2012.

[18] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study on factors impacting bug fixing time," 2012 19th Working Conference on Reverse Engineering, 2012.

[19] G. Pinto, M. Reboucas, and F. Castor, "Inadequate testing, time pressure, and (over) confidence: A tale of continuous integration users," 2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), 2017.

[20] G. Pinto, F. Castor, R. Bonifacio, and M. Rebouças, "Work practices and challenges in continuous integration: A survey with Travis CI Users," Software: Practice and Experience, vol. 48, no. 12, pp. 2223–2236, 2018.

[21] G. Sizilio Nery, D. Alencar da Costa, and U. Kulesza, "An empirical study of the relationship between continuous integration and test code evolution," 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2019.

[22] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at Google)," Proceedings of the 36th International Conference on Software Engineering, 2014.

[23] I. Keskin Kaynak, E. Çilden, and S. Aydin, "Software quality improvement practices in continuous integration," Communications in Computer and Information Science, pp. 507–517, 2019.

[24] J. Pearl, Causality: Models, reasoning, and inference. Cambridge: Cambridge University Press, 2005.

[25] J. Textor, B. van der Zander, M. S. Gilthorpe, M. Liśkiewicz, and G. T. H. Ellison, "Robust causal inference using directed acyclic graphs: The R package 'dagitty,"' International Journal of Epidemiology, 2017.

[26] K. Beck, Extreme programming explained: Embrace change. Boston: Addison-Wesley, 2010.

[27] L. D. Panjer, "Predicting eclipse bug lifetimes," Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007), 2007.

[28] M. A. Hernan and J. M. Robins, Causal inference. Boca Raton, FL: CRC, 2010.

[29] M. C. de Oliveira, "Draco: Discovering refactorings that improve architecture using fine-grained co-change dependencies," Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017.

[30] M. C. de Oliveira, D. Freitas, R. Bonifácio, G. Pinto, and D. Lo, "Finding needles in a haystack: Leveraging co-change dependencies to recommend Refactorings," Journal of Systems and Software, vol. 158, p. 110420, 2019.

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

[32] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution," Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018.

[33] Mockus and Votta, "Identifying reasons for software changes using historic databases," Proceedings International Conference on Software Maintenance ICSM-94, 2000.

[34] M. R. Islam and M. F. Zibran, "Insights into continuous integration build failures," 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017.

[35] M. Reboucas, R. O. Santos, G. Pinto, and F. Castor, "How does contributors' involvement influence the build status of an open-source software project?," 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017.

[36] N. Cartwright,"Nature's Capacities and Their Measurement," pp. 141–182, 1994.

[37] N. Cassee, B. Vasilescu, and A. Serebrenik, "The silent helper: The impact of continuous integration on code reviews," 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020.

[38] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," 2014 IEEE International Conference on Software Maintenance and Evolution, 2014.

[39] P. M. Duvall, S. Matyas, and A. Glover, Continuous integration improving software quality and reducing risk. Upper Saddle River, NJ: Addison-Wesley, 2013.

[40] P. Spirtes, C. N. Glymour, and R. Scheines, Causation, prediction, and Search. Cambridge, MA: MIT Press, 2000.

[41] S. A. Licorish and S. G. MacDonell, "Exploring software developers' work practices: Task differences, participation, engagement, and speed

of Task Resolution," Information & Management, vol. 54, no. 3, pp. 364–382, 2017.

[42] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software," 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017.

[43] T. Zhang, H. Jiang, X. Luo, and A. T. S. Chan, "A literature review of research in bug resolution: Tasks, challenges and future directions," The Computer Journal, vol. 59, no. 5, pp. 741–773, 2015.

[44] Vasilescu B, Yu Y, Wang H, et al (2015) Quality and productivity outcomes relating to continuous integration in GitHub. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015. doi: 10.1145/2786805.2786850

[45] V. Garousi and M. V. Mäntylä, "When and what to automate in software testing? A multi-vocal literature review," Information and Software Technology, vol. 76, pp. 92–117, 2016.

[46] W. Felidre, L. Furtado, D. A. Costa, B. Cartaxo, and G. Pinto, "Continuous Integration Theater," 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2019.

[47] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The impact of continuous integration on other software development practices: A large-scale empirical study," 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017.