- doi:10.1109/AST58925.2023.00011
- https://ieeexplore.ieee.org/abstract/document/10174236

SourceWarp: A scalable, SCM-driven testing and benchmarking approach to support data-driven and agile decision making for CI/CD tools and DevOps platforms

Julian Thome[†*], James Johnson[‡], Isaac Dawson[†§], Dinesh Bolkensteyn[†¶], Michael Henriksen[†‖], Mark Art[†**]
[†]GitLab Inc.
[*]jthome@gitlab.com, [‡]jamxjohn@gmail.com, [§]idawson@gitlab.com, [¶]dbolkensteyn@gitlab.com [‖]mhenriksen@gitlab.com
[**]mart@gitlab.com

***Abstract*—The rising popularity and adoption of source-code management systems in combination with Continuous Integration and Continuous Delivery (CI/CD) processes have contributed to the adoption of agile software development with short release and feedback cycles between software producers and their customers. DevOps platforms streamline and enhance automation around source-code management systems by providing a uniform interface for managing all the aspects of the software development lifecycle starting from software development until software deployment and by integrating and orchestrating various tools that provide automation around software development processes such as automated bug detection, security testing, dependency scanning, etc..**

Applying changes to the DevOps platform or to one of the integrated tools without providing data regarding its real world impact increases the risk of having to remove/revert the change. This could lead to service disruption or loss of confidence in the platform if it does not perform as expected. In addition, integrating alpha or beta features, which may not meet the robustness of a finalised feature, may pose security or stability risks to the entire platform. Hence, short release cycles require testing and benchmarking approaches that make it possible to prototype, test, and benchmark ideas quickly and at scale to support Data-Driven Decision Making, with respect to the features that are about to be integrated into the platform.

In this paper, we propose a scalable testing and benchmarking approach called *SourceWarp* that is targeted towards DevOps platforms and supports both testing and benchmarking in a cost effective and reproducible manner. We have implemented the proposed approach in the publicly available *SourceWarp* tool which we have evaluated in the context of a real-world industrial case-study. We successfully applied *SourceWarp* to test and benchmark a newly developed feature at GitLab which has been successfully integrated into the product. In the case study we demonstrate that *SourceWarp* is scalable and highly effective in supporting agile Data-Driven Decision Making by providing automation for testing and benchmarking proof-of-concept ideas for CI/CD tools, chained CI/CD tools (also referred to as pipeline), for the DevOps platform or a combination of them *without* having to deploy features to the staging or production environments.

***Index Terms*—Software Testing, Test automation, Benchmarking, Data-driven decision making**

I. Introduction

Agile software development is widely adopted in the software industry: a recently conducted survey from KPMG with 120 companies from 17 countries revealed that 70% of them have already shifted or are in the process of shifting towards Agile. The main motivation for this shift is product delivery at a higher speed while increasing customer satisfaction [1].

The rising popularity and adoption of source-code management systems (SCM) in combination with Continuous Integration and Continuous Delivery (CI/CD) processes have contributed to the adoption of agile software development [2]. While SCMs enable developers to track code changes, maintain a history of these changes and roll-back or revert changes, CI/CD provides automation and tooling around the integration and deployment of code changes (e.g., compilation, syntax checks, compliance checks, execution of unit-tests, security tests, etc.). The core idea of CI/CD is to provide automation around software/source-code managed by an SCM. On industrial systems, the automation is realized by means of CI/CD tools which are often implemented as micro-services and shipped/deployed as Docker[1] images. CI/CD tools can be used to automatically check source code for errors, run automated builds, run unit tests, changelog and version management and to automatically deploy the software to production.

CI/CD practices build the foundation of DevOps; DevOps is a neologism created from the two words **Dev**elopment and **Op**eration**s** combining both the process of software development with the operations or automation around the software development process. DevOps is a software development philosophy commonly used in the software industry that aims to streamline the process of software-development by maintaining a high velocity with which they are shipped and providing short feedback cycles for customers. These short feedback cycles can be used to monitor the impact of a feature from the point where it is shipped and inform developers and product managers about the success or failure of a given deployment

[1]https://www.docker.com/

which can then be used to make data-driven decisions about feature integration, deprecation or improvement. This Data-Driven Decision Making (DDDM) process, i.e., the process of validating a feature before making a decision about its integration, is a crucial part of agile software development in order to systematically and rapidly evolve a software product or service.

A DevOps platform manages and orchestrates the execution, in and output of the various CI/CD tools and provides an interface to realize user-facing functionality. Figure 1 provides a simplistic birds-eye view of the anatomy of a DevOps platform. Macro-blocks are illustrated as dashed boxes; components belonging to the macro-block are depicted as squared boxes and processes are depicted as round-boxes. Input/output relationships are depicted as arrows. A DevOps platform can host multiple projects each of which is connected to an SCM repository. Automatic or manual actions that are performed on the SCM repository (such as pushes, commits, scheduled events etc.) can trigger CI/CD jobs that run CI/CD tools which usually rely on the data from the SCM repository as input.

For example, a developer may push changes to a Git repository; this action initiates four jobs running CI/CD tools each of which is responsible for performing security code analysis, dependency scanning, a unit test and an automated build. The result from these jobs (security reports, test reports, etc.) are then digested by the backend service and stored in the backend database which persists and tracks all the data produced by the CI/CD tools over time.

As depicted in Figure 1, CI/CD tools can also have input/output relations between themselves; this setup is used in situations where one CI/CD tool verifies or uses the data produced by another one as input. This is useful in the context of security where a SAST CI/CD tool verifies, augments or refines the results produced by another SAST CI/CD tool.

The results from the CI/CD tools are then digested by the backend service from the DevOps platform before being stored in a backend database. DevOps platforms usually also expose an Application Programming Interface (API) as another means to interact with the platform and its components.

The data available in the backend database can then be used to realize user-facing functionality (by means of a user interface); the examples from Figure 1 include:

- *Vulnerability Management* shows reported vulnerabilities from potentially multiple static application security testing (SAST) CI/CD tools whose results are stored in a backend (vulnerability) database so that users of the platform are able to act on them by prioritizing, dismissing or possibly fixing them.
- *Dependency Management* provides insights about the software supply chain and used software dependencies.
- *License Compliance* shows all licenses (including the ones from 3rd party dependencies) and highlights potential incompatibilities between them.
- *Code Quality* notifies developers about code quality issues or deviation from best practices.
- *Audit Events* provides insights about important events (e.g, release creation, merge/pull request update) through an audit log.
- *CI/CD status* provides an overview of succeeded/failed jobs and access to the corresponding logs.

As illustrated in Figure 1, DevOps platforms are heterogeneous systems that consist of many components; they act as an integration point for different micro-services which we refer to as CI/CD tools that contribute to some of the user-facing functionality. For example, from a user perspective *Vulnerability Management* is visible as a single functionality while the data for realizing it may have been produced by a chain of different CI/CD tools analyzing different states of the SCM repositories over time and storing them in the backend database while keeping track of user-interactions on those findings that took place through the user interface or the API.

Hence, the integration of any new features or applying any changes to the DevOps platform or on one of the integrated CI/CD tools is challenging and risky because any functional or performance bug that may be introduced could have a negative cascading effect potentially impacting the whole DevOps platform. Having preliminary data about the impact of a feature is important to partially mitigate these risks by providing quantifiable evidence that the feature provides value.

This leads to a chicken-or-egg problem: a feature, albeit a small one, has to be deployed or shipped before its impact can be measured. DDDM would have required the data to be collected in order to assess the value of the feature in the first place. Shipping a feature without any data about its impact entails the risk of having to remove it if it does not perform as expected. In addition, (partially) integrating proof-of-concepts (PoCs) and, thus, not necessarily robust product features may increase risks concerning the stability and security of the product. To mitigate these risks, developers and product managers may limit the velocity with which new features are developed and deployed which goes against the Agile philosophy.

Commonly applied software best practices such as the use of staging environments, feature flags behind which features can be gated, or testing strategies such as canary testing approaches [3], [4] proved to be useful in partially addressing the stability and potential security issues; variant testing approaches such as A/B testing [5], [6] provide valuable insights in order to compare deployed (mostly user-facing) feature variants. However, these approaches are not designed to provide preliminary insights with regards to the impact of a feature *before* it is deployed to staging and/or production environments in order to support DDDM. In addition, A/B and canary testing approaches are usually not deeply integrated with SCM systems for the purpose of test-data extraction which is important for benchmarking and testing heterogeneous DevOps platforms that are centered around code and code-changes tracked by SCM systems.

In this paper we present *SourceWarp*, a scalable, SCM-driven benchmarking and testing framework targeted towards

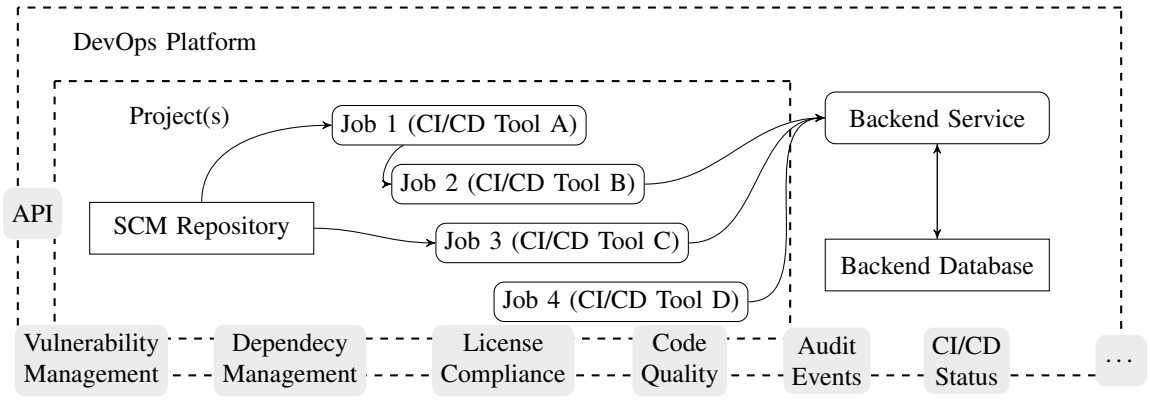

Fig. 1. Anatomy of DevOps platform. This figure illustrates a simplistic view of a DevOps platform. Macro-blocks are illustrated as dotted boxes; components belonging to the macro-block are depicted as squared boxes and processes are depicted as round boxes. Input/output relationships are depicted as arrows. A DevOps platform can host multiple projects each of which is connected to an SCM repository. Actions (such as pushes, commits, scheduled events etc.) can trigger jobs. A job runs a CI/CD tool which usually operates on the data from the SCM Repository. CI/CD tools can also have input/output relations between themselves. The results from the CI/CD tools are then digested by a backend service from the DevOps platform before being stored in a backend database. DevOps platforms usually also expose an Application Programming Interface (API) as another means to interact with the platform and its components. In addition, DevOps platforms provide user-facing functionality that leverage, present and enable users to interact with the data produced by the various CI/CD jobs in a unified and user-friendly way. Some examples such as Vulnerability Management, Dependency Management, etc. are displayed at the bottom of the figure.

DevOps platforms and CI/CD tools. *SourceWarp* enables developers and product managers to run and assess the results of DDDM experiments on DevOps platforms and CI/CD tools *without* having to deploy the feature to production and, thus, does not limit the velocity with which developers can prototype and try out new feature ideas. Being solely dependent on the SCM for the purpose of sourcing test-data, the proposed *SourceWarp* approach can be used to quickly (re-)run experiments parallel to the feature development which is especially useful in an agile context.

We present a real-world industrial case-study for which we successfully applied *SourceWarp* to test and benchmark a newly developed feature at GitLab and which has been successfully integrated into the product. In the case study we demonstrate that *SourceWarp* is scalable and highly effective in supporting DDDM by providing automation for testing and benchmarking PoCs on large code bases.

II. Data-driven Decision Making in the context of DevOps platforms in an industrial setting

In order to effectively assess the effects introduced by new features or changes applied to CI/CD tools and platforms, which we also refer to as *system*, *SourceWarp* addresses the following challenges:

- *Observability* As illustrated in section I, DevOps tools usually rely on the data from SCM which is constantly changing. Hence, if we want to collect meaningful data, we have to take into consideration the different states that the SCM can be in so that we have to observe and collect multiple data-points over time. For example, if we want to collect meaningful data about the efficacy of two SAST CI/CD tools, just knowing the findings they produce on a test project is not sufficient to make a decision about which one of them is better as it does not take into consideration the progression of the criterion *Number of Vulnerabilities* as the test project evolves and vulnerabilities are continuously added to the backend database.
- *Reproducibility* An agile DDDM process has to be incremental in the sense that we have a baseline system to compare against an updated system which we refer to as PoC. This, however, implies that we have to be able to run, reproduce the same tests on different systems, i.e., the baseline and the changed systems so that we can correlate the extracted data/metrics.
- *Scalability* The code-bases we work with on a daily basis are fairly large (MLOC). Using SCMs that host large code-bases with large histories poses challenges from a scalability standpoint.
- *Configurable Granularity/Resolution* For our use-case it is important that our testing approach actively supports cherry-picking test-data and configuring the granularity with which the tests are executed, the benchmarking data and metrics are collected. This is a mechanism for balancing/trading-off Observability and Scalability.

Our DDDM process when looking at the data provided by *SourceWarp* as part of a benchmarking and testing experiment on a PoC is mainly guided by the three questions below. If all of the questions below can be answered positively, the change is ready to be deployed to the staging or production environments.

- *Performance Overhead* Is the overhead of the updated system (in terms of execution time) negligible?
- *Improvement* Considering the collected data, can we observe an improvement when comparing the baseline system against the updated system?
- *Robustness* Is the updated system free from any crashes?

III. Approach

The *SourceWarp* approach is depicted in Figure 2. Processes are depicted as rounded boxes, whereas input/output artifacts are depicted as squared boxes. The approach starts with the SCM source code repository as input and produces a results/metrics report which covers the testing and benchmarking results. *SourceWarp* consists of three main phases each of which is explained in detail in a corresponding subsection.

(A) The *Record* phase extracts a subset of commits that have been applied to the source code repository.

(B) The *Patch Sequence Generation* phase prepares the patch sequence to be applied to the target SUT which is represented by the area in Figure 2 that is labeled as *System under Test (SUT)*. In our use case, a SUT can be a CI/CD tool, a chain of CI/CD tools (also referred to as pipeline), a DevOps platform(s) or a combination of them.

(C) During the *Monitor phase* the actual patches are applied and the behaviour of the target SUT is observed and evaluated.

SourceWarp relies on a version-controlled SCM repository where the history of the repository itself holds the test-data. Since *SourceWarp* leverages a source-code repository to hold the test-input data, it is predestined for (but not limited to) replaying histories of source-code projects which is particularly useful for applying DDDM to test and assess newly developed features for DevOps platforms or CI/CD tools as these tools usually operate on data provided by SCMs.

In a record phase, *SourceWarp* extracts commits from the source history that are relevant with respect to a given test-criterion and generates a patch replay sequence. In the replay phase, *SourceWarp* replays the generated sequence on a target SUT. While replaying commits from the project history, *SourceWarp* captures results/metrics by executing evaluation scripts in order to capture timing information, API responses, files, etc. from the Systems under Test.

In the context of applying DDDM with regards to a newly implemented feature, *SourceWarp* can be used to apply an automated form of variant testing by feeding data that has been recorded in the record phase to the original, unchanged SUT, and to a modified SUT that includes the feature to be evaluated. *SourceWarp* evaluates both systems by replaying the same source history to both of them and by capturing the results and metrics that are computed by configurable evaluation scripts.

The collected data can be used for ...

1) ... DDDM by helping developers and product managers to better understand and quantify the impact of newly developed features for a DevOps platform or a CI/CD tool before a potential product integration.
2) ... spotting and identifying bugs/regressions while replaying the events from the project history.

For the remainder of this section, we assume that an SCM repository can be represented as an ordered set of commits, i.e., the project history H as defined below.

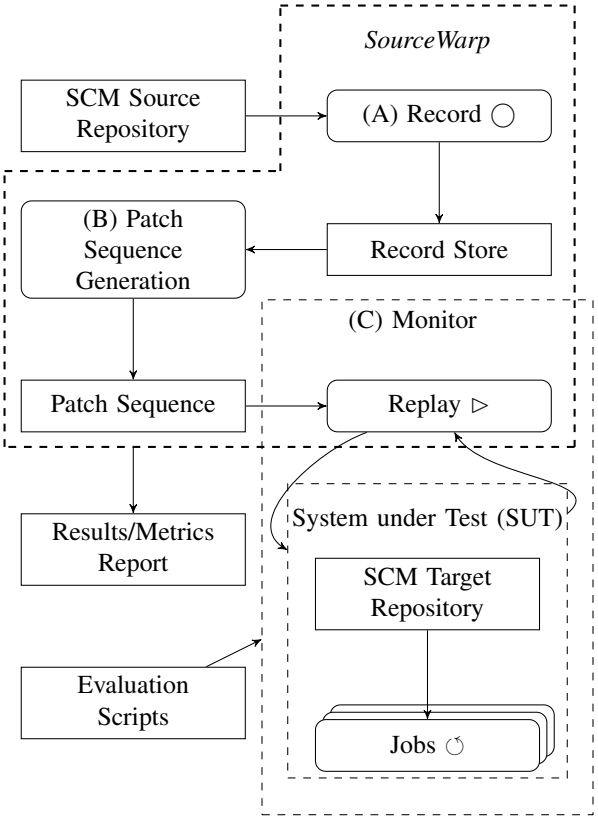

Fig. 2. The *SourceWarp* approach. The arrows represent input/output relations; the dashed boxes represent macro-operations/steps; rounded boxes represent (sub-)operations/steps, squared boxes represent input/output artifacts. The *Record* step extracts source code changes from the SCM source repository and stores them in a Record Store; the *Patch Sequence Generation* step samples a patch sequence which is then replayed on a system under test while the system is monitored. The *Monitor* step ensures that the patch sequence can be successfully replayed on the SUT which may harbour an SCM target repository; the SUT may spawn additional (CI/CD) services. The monitor steps observes whether the history can be replayed without uncovering erroneous behaviour on the SUT while collecting metrics that can be used for DDDM. The metrics are collected by invoking user-defined scripts.

$$H = \{c_1, c_2, \ldots, c_n\}$$

c_1 denotes the first and c_n the last commit and n denotes the index number of the last commit. Every commit has a time-stamp $t(c_x)$ that marks the point in time at which it has been included into H with $t(c_x) \leq t(c_{x+1})$.

A. Record

The record phase extracts merge request commits from the history H of the SCM source repository based on a specified time interval T as defined below.

$$T = [T_{min}, T_{max}]$$

T is framed by T_{min} as start time and T_{max} as end time. T can be used to extract commits that happened within a certain time frame which are then included in the time slice S' as defined below.

$$S' = \{c_s | c_s \in H \wedge T_{min} \leq t(c_s) \leq T_{max}\}$$

For the purpose of focusing the recorded information on files that are relevant with respect to the behaviour evaluated/tested by *SourceWarp*, it is possible to provide an allow-list that includes the names of the files to focus on so that we would obtain the time slice S as defined below.

$$S = \{c_s | c_s \in S' \wedge \text{allow-list}(c_s)\}$$

We assume that allow-list(c_s) evaluates to true if the commit c_s is related to a file that is specified in the allow-list. If no allow-list is provided, allow-list(c_s) always evaluates to true. It is important to note that the order of the commits is always preserved so that $t(c_s) \leq t(c_{s+1})$ always holds true.

Both the time frame and the allow-list are used for reducing replay time as well as focusing the analysis on specific files which enables *SourceWarp* to scale to large repositories both in terms of history size and number of files stored in the repository. The slice S is then persisted in the record store so that it can be replayed.

B. Patch Sequence Generation

Based on the slice $S = \{c_1, c_2, c_3, \ldots, c_m\}$, the patch sequence generation is responsible for generating a sequence of patches. A patch can be considered as a code change that is going to be applied atomically on the target SUT. *SourceWarp* makes it possible to define a patch sampling number N, i.e., the number of commits that should be combined into a single patch. Hence, the patch sequence can be represented as a partition from the slice S as illustrated in the example below.

$$PS = \bigcup_{i=1}^{|S| \div N} \text{patch}_i \text{ with } \text{patch}_i = \bigcup_{k=(i-1)N+1}^{iN} c_k \text{ and } c_k \in S$$

For the example slice $S = \{c_1, c_2, c_3, c_4, c_5, c_6\}$ with a patch sampling number of $N = 2$, we would obtain the patch sequence $PS = \{\{c_1, c_2\}, \{c_3, c_4\}, \{c_5, c_6\}\}$ where each of the inner sets represents a patch. For example, $\text{patch}_1 = \{c_1, c_2\}$ includes two commits that are applied atomically on the target SUT.

Apart from the time window T and the allow-list, the patch sampling number N is another control that is aimed to increase scalability of *SourceWarp*. T and the allow-list are targeted towards reducing the time spent in the record phase and enable *SourceWarp* scale to SCM's with a large history. However, these heuristics have an impact on the time spent in the replay phase too because the smaller $|S|$, the less patches have to be replayed; the patch sampling number N is focused on reducing the replay time spent within the heterogeneous DevOps platform.

C. Monitor

The Monitor phase leverages the patch sequence by replaying it on the target SUT. The target SUT can be a CI/CD tool, a chain of CI/CD tools (also referred to as pipeline), a DevOps platform(s) or a combination of them. As depicted in Figure 2, the Monitor phase consists of a replay step that replays patches to the SUT. *SourceWarp* does not make any assumptions about the SUT; it only requires the SUT to be accessible by means of an API that enables us to ...

1) ... wait for the execution of services/actions that may be triggered when applying a patch from PS.
2) ... extract the relevant test/evaluation data.

However, due to the variety of systems on which *SourceWarp* could be applied as well as the variety of data-points to be collected, we assume that the API communication and the data-extraction for the SUT is provided in the form of evaluation scripts as part of the *SourceWarp* configuration.

SourceWarp natively supports the presence of SCM target repositories in the SUT which may spawn additional (CI/CD) jobs all of which are considered as belonging to the SUT itself. In the presence of an SCM target repository, before starting to replay the patch sequence PS, *SourceWarp* initializes a new SCM target repository with the history $H' = \{c_1, c_2, \ldots, c_x\}$ based on the history of the original SCM source repository $H = \{c_1, c_2, \ldots, c_n\}$ with $H' \subseteq H$ and $t(c_x) < T_{min} \leq t(c_n)$ on the target SUT.

SourceWarp also supports overwriting certain files from H' so that they can be directly passed-through the target SUT. This is useful to handle situations where the code that is stored in the repository itself has an impact on the replay time. In CI/CD environments, it is common that the CI/CD configuration itself is stored in the SCM. However, it may not always be required to run the entire set of CI/CD jobs. If a *SourceWarp* is used to evaluate a particular job that is part of a larger CI/CD deployment it is sufficient to run the job in isolation which can be controlled by using a custom CI/CD with which the standard configuration could be overwritten.

The record phase then leverages the patch sequence PS by replaying and applying every single patch iteratively on the target SUT through the API. In the presence of an SCM target repository, the patches may be submitted to the SCM target repository directly. After the application of a patch, the monitor phase monitors the effect of applying the patch file. As mentioned earlier, this step ...

1) ... ensures that the SUT works as expected (*Testing*).
2) ... extracts/collects results/metrics from the SUT while applying the patches by invoking user-defined evaluation scripts (*Benchmarking*).

For example, an evaluation script can collect data from an API that is exposed by the SUT or simply parse a JSON vulnerability report that was generated by the SUT and count the number of reported vulnerabilities.

To avoid incorrect interpretation of the results yielded in the monitoring phase, the *SourceWarp* approach is usually applied to test the impact of a single feature in isolation: if multiple, chained CI/CD tools perform a task that is relevant with respect to the tested feature, they can be treated as a single system with a final output which we could then use for both testing and benchmarking.

IV. Industrial Case Study: Vulnerability Tracking

This section describes an actual internal evaluation we performed with *SourceWarp* to support DDDM for a newly developed *Vulnerability Tracking* feature which has since been successfully integrated into GitLab[2].

GitLab relies on a heterogeneous static application security testing (SAST) setup where multiple SAST tools are executed in combination to detect as many security issues of different kinds and natures as early as possible in the software development lifecycle for the managed software projects. SAST (DevOps) tools are integrated as Docker images and communicate findings via the backend services to the backend (vulnerability) database as depicted in Figure 1. The findings are then available through the user-facing Vulnerabilty Management functionality.

Vulnerabilty Management is the semi-manual process of interacting with the vulnerabilities that are stored in a backend (vulnerability) database by classifying, prioritizing, dismissing and/or possibly fixing them. Vulnerabilty Management is a semi-manual process and for this reason it is important to minimize the amount of noise (e.g., redundant vulnerability findings) presented to the user/security analyst in order to reduce manual auditing effort.

However, when applying Vulnerabilty Management in combination with a heterogeneous SAST setup and SCMs, there are two potential sources of noise that can lead to the duplication of findings and, thus, to an increased manual auditing time during the Vulnerabilty Management phase:

1) *Double Reporting*: In a heterogeneous SAST multiple tools might report the same findings.
2) *Code Volatility*: The location of vulnerabilities is constantly changing. Deciding whether a vulnerability in one version of a software project is the same as a previously detected vulnerability cannot be done without a clear definition of what makes one vulnerability distinct from another: source code may be added, removed, edited, shifted, auto-formatted, etc. as the project grows.

Vulnerability Tracking is an automated process that helps deduplicating and tracking vulnerabilities throughout the lifetime of a software project. At GitLab, we have implemented a new Vulnerability Tracking approach initially as a proof-of-concept (PoC) to reduce the negative effect (noise) of both double reporting as well as code volatility. Before the integration of the new Vulnerability Tracking approach, we have been using a line-based approach which identified (*fingerprinted*) findings based on the file and the line number on which they appeared. Hence, line-based fingerprinting served as the baseline we aimed to improve.

Figure 4 shows how the Vulnerability Tracking approach worked as a CI/CD tool. Artifacts and databases are depicted as squared boxes whereas rounded boxes represent processes; edges represent input/output relations. The source code is managed through an SCM; one or multiple SAST CI/CD tools check the source-code for vulnerabilities and generate vulnerability reports which are then post-processed by means of a Vulnerability Tracking post-processor that computes fingerprints, i.e., hashes that uniquely identify vulnerabilities. The fingerprints are required for both deduplication and vulnerability Tracking. The post-processor generates a refined vulnerability report that includes a set of deduplicated vulnerabilities with their fingerprinting information. The data is then stored in a backend (vulnerability) database, which is part of the DevOps platform, that collects and stores all the data used for the purpose of Vulnerabilty Management. This example illustrates that the systems under test in the context of DevOps platforms are often heterogeneous. In this particular example, we have multiple dependant CI/CD tools whose execution is orchestrated by means of the DevOps platform that also stores the output/results produced by the CI/CD tools.

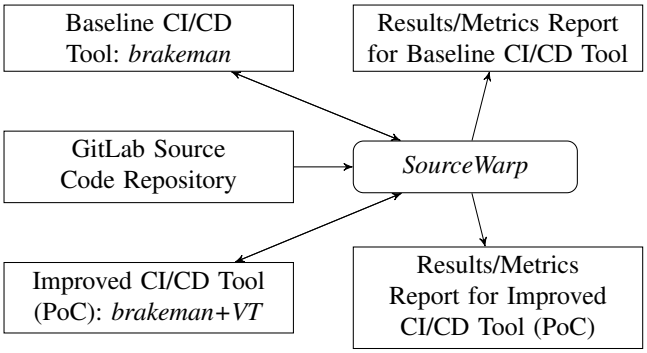

Fig. 3. Evaluation Setup: Boxes denote input/output artifacts, arrows denote input/output relations. *SourceWarp* uses the GitLab source repository as input from which it replays a slice of the history on two target systems under test which were both provided as Docker images: *brakeman* and *brakeman+VT*, i.e., the baseline system that used the brakeman SAST tool without any changes and the improved tool that included the new Vulnerability Tracking feature, respectively. As output *SourceWarp* generated two reports for both systems under test which we used to evaluate the impact of *brakeman+VT*

However, before even considering to integrate the new Vulnerability Tracking method into the product we had to collect evidence that the PoC performed better than the baseline. We used *SourceWarp* for solving this DDDM problem. This case study aims to address the following research questions:

- *RQ1*: Is *SourceWarp* effective in supporting DDDM?
- *RQ2*: What is the impact of the allow-list and the patch sampling number?

A. Evaluation Setup

As SCM we used the GitLab source code Git repository[3] because it is a real-world repository hosting a large codebase. In addition, the PoC, should it be successfully evaluated, would be initially deployed on the GitLab code repository which made it a natural choice for the initial setup.

In February 2021, which was the year when we evaluated the PoC, GitLab just released v13.6.6-ee which included 34K

[2]https://docs.gitlab.com/ee/user/application_security/sast/#advanced-vulnerability-tracking

[3]https://gitlab.com/gitlab-org/gitlab

TABLE I
EVALUATION RESULTS WHEN RUNNING *SourceWarp* ON OUR TWO SYSTEMS UNDER TEST: *brakeman* AND OUR IMPROVED PROTOTYPE *brakeman+VT*

	Recording Time	Replay Time	Average Replay Time Per Patch	Overall Time	↯	#Unique Fingerprints for patch$_i$ 1	2	3	4	5	6	7	8	9	10	11	12	13
brakeman	54 min 30 s	18 min 19 s	1 min 24 s	1 h 12 min 49 s	0	94	94	97	97	102	102	118	118	125	125	128	128	132
brakeman+VT	54 min 30 s	17 min 50 s	1 min 22 s	1 h 12 min 20 s	0	83	83	84	84	84	84	91	91	91	91	91	91	92
Δ_{abs}	0 s	29 s	2 s	29 s	0	11	11	13	13	18	18	27	27	34	34	37	37	40
$\Delta_{\text{rel}}(\%)$	0	2.6	2.6	0.6	0	11	11	13	13	17	17	22	22	27	27	28	28	30

Columns *Record Time*, *Replay Time* display the time *SourceWarp* took during the record and replay phases, respectively; column *Average Replay Time Per Patch* shows how much time *SourceWarp* spent on average to replay a single patch; column ↯ displays the number of crashes; column *#Unique fingerprints for patch$_i$* shows the number of unique vulnerability findings observed after replaying patch$_i$ where the patch sequence numbers are provided in the row below. Row *brakeman* shows all the results that were related to the unchanged/original brakeman SAST tool we used at the time with line-based fingerprinting; row *brakeman+VT* shows the results for our SAST PoC that used the enhanced Vulnerability Tracking algorithm. The rows Δ_{abs} and Δ_{rel} display absolute and relative difference expressed in percentage (improvement).

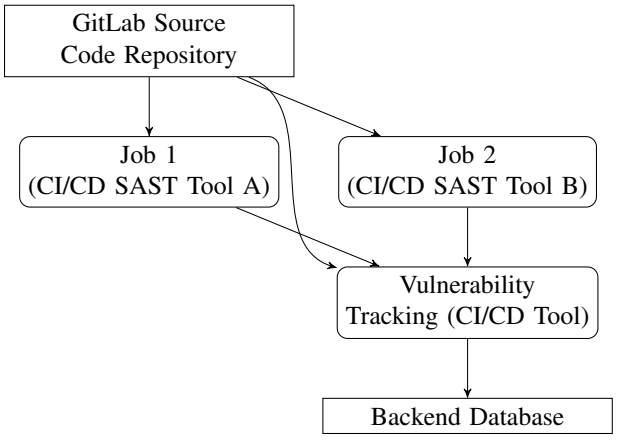

Fig. 4. This figure depicts how Vulnerability Tracking works with existing SAST CI/CD tools. Artifacts and databases are depicted as squared boxes whereas rounded boxes represent processes; edges represent input/output relations. The source code is managed through an SCM; one or multiple SAST scanners check the source-code for vulnerabilities and generate vulnerability reports which are then post-processed by means of a Vulnerability Tracking CI/CD tool that computes fingerprints, i.e., hashes that uniquely identify vulnerabilities. The fingerprints are required for both deduplication and Vulnerability Tracking. The post-processor generates a refined vulnerability report that includes a set of deduplicated vulnerabilities with their fingerprinting information. The data is then stored in a backend (vulnerability) for the purpose of Vulnerabilty Management.

files, with 3.7 million lines of code out of which 1.2 million lines of code were Ruby on Rails (RoR) code with a history of 200K commits.

The evaluation setup is illustrated in Figure 3. *SourceWarp* used the GitLab source repository as input from which it replayed a slice of the history on two target systems under test both of which were provided as Docker images: *brakeman* and *brakeman+VT*, i.e., the baseline system (CI/CD tool) that used the brakeman SAST tool without any changes and the changed system (CI/CD tool) that included the new Vulnerability Tracking feature, respectively. As output *SourceWarp* generated two reports for both systems under test which we used to evaluate the PoC. For our experiment we used two systems under test:

1) As baseline we used the dockerized SAST tool for RoR GitLab brakeman v2.12.2[4] which wraps the free and open source RoR SAST tool brakeman[5]. GitLab brakeman v2.12.2 was the most recent release in February 2021 when we performed the experiment and which we refer to as *brakeman*.
2) As our PoC we used a dockerized version of brakeman v2.12.2 enhanced with our new Vulnerability Tracking approach and which we refer to as *brakeman+VT*.

We leveraged the integrated *SourceWarp* support for Docker since all analyzers were already available as Docker images: all SAST tools are provided in the form of Docker images per default as per the automated build process GitLab uses for all product-related CI/CD tools. We arbitrarily picked the time window between 2020-10-31 and 2020-12-31 as T_{min} and T_{max}.

As allow-list, we provided a list of 60 relevant files for this time frame. This information was downloadable through GitLab's vulnerability management system that provides a historical record of all vulnerabilities that have been detected by all SAST tools. The experiment has been conducted on Linux Laptop (using version 5.10.12) with a Core i7 hexa-core (2.2GHz per core) with an Intel SSD Pro 7600p using Docker 20.10.3.

B. RQ1: Effectiveness of SourceWarp for supporting DDDM

With respect to DDDM, we benchmarked the newly implemented Vulnerability Tracking PoC by running *SourceWarp* with evaluation scripts for collecting data in order to assess the criteria listed below, which are all derived from the process outlined in section II, and to make a decision about product integration. Note that the development of the evaluation scripts for this experiment amounted to $\sim 2\,\text{h}$.

1) *Overhead*: What is the overhead (in terms of execution time) when applying the new Vulnerability Tracking method. The overall execution time is the sum of the record time and replay time.
2) *Cost savings*: Auditing effort denotes the time that has to be put into manually verifying a finding, dismissing

[4]https://gitlab.com/gitlab-org/security-products/analyzers/brakeman
[5]https://brakemanscanner.org/

and/or fixing it. Our hypothesis being that the new Vulnerability Tracking method reduces manual auditing effort during Vulnerability Management, we want to know an estimate by which margin the auditing time could be reduced with the new method as opposed to the old, line-based method.

3) *Robustness*: Given that the new Vulnerability Tracking would be ultimately deployed to all integrated SAST scanners, we wanted to make sure that their PoC implementation is free from any crashes.

Table I shows the data included in the result/metrics report produced by *SourceWarp* for both *brakeman* and *brakeman+VT*. Column *Record Time* displays the time *SourceWarp* took during the record phase; column *Replay Time* shows the time *SourceWarp* required to replay all the patches in the patch sequence; column *Average Replay Time Per Patch* shows how much time *SourceWarp* spent on average to replay a single patch; column ↯ displays the number of crashes *SourceWarp* observed during the record phase and column *#Unique fingerprints for patch*$_i$ shows the number of unique vulnerability findings observed after replaying patch$_i$ where the patch sequence numbers are provided in the row below. Note that for *#Unique fingerprints for patch*$_i$ smaller numbers denote an improvement: the less unique fingerprints are generated, the better the PoC performed with regards to deduplication. The *brakeman* row shows all the results that were related to the unchanged/original brakeman SAST tool we used at the time with line-based fingerprinting; the *brakeman+VT* row shows the results for our SAST PoC that used the enhanced Vulnerability Tracking algorithm. The rows Δ_{abs} and Δ_{rel} display absolute and relative difference expressed in percentage (improvement) between *brakeman+VT* and the baseline *brakeman*. As already mentioned in section II for DDDM it is important to collect multiple data-points during the Replay phase in order to properly reflect and observe the progression in terms of the number of vulnerabilities that accumulate in the backend (vulnerability) database over time to which ultimately a user of the system would be exposed.

The recording phase, which was only performed once, took the same amount of time ($54\,\text{min}\ 30\,\text{s}$) for both *brakeman* and *brakeman+VT*. The replay time was almost the same in both cases; in fact *brakeman+VT* was $29\,\text{s}$ (2.6%) faster which was due to the overhead incurred when creating the Docker containers. From the fact that the Vulnerability Tracking overhead has been shadowed by Docker container creation time, we can conclude that the overhead of the new Vulnerability Tracking algorithm is negligible in practice. Table I shows that no crashes have been observed during the replay phase. In addition, we can see that the *brakeman+VT* performed significantly better than *brakeman* in terms of deduplicating findings. After the application of the first patch *patch*$_1$, *brakeman+VT* deduplicated 11 vulnerabilities; for every subsequently applied patch the number Δ_{abs} of successfully eliminated, deduplicated, redundant findings when using *brakeman+VT* constantly increases reaching 40 after the application of the *patch*$_{13}$ which is a relative improvement of 30%.

These results served as an input for DDDM: with a negligible overhead, we achieved cost savings of 30%, i.e., reduction of curation/auditing effort, without any negative impact in terms of robustness (no crashes observed).

SourceWarp enabled us to perform this experiment in a fully automated and reproducible manner and provided all the required data to make an informed decision about the product integration of Vulnerability Tracking. Concerning *RQ1*, *SourceWarp* proved to be highly effective for the purpose of DDDM.

C. RQ2: The impact of the allow-list and the patch sampling number

The purpose of allow-list and patch sampling numbers is to make *SourceWarp* scale to large code repositories and to be able to quickly (re-)run experiments during feature development due to the reduction in terms of execution time which is discussed in more detail below.

Impact of the allow-list

We ran *SourceWarp* on a slice of the git history staring from 2020-10-31 until 2020-12-31 with a slice size of $|S'| = 2748$. As depicted in Table I, the average replay time ranged between $1\,\text{min}\ 22\,\text{s}$ and $1\,\text{min}\ 24\,\text{s}$ so that, if we are assuming the average replay time per patch to be $1\,\text{min}\ 20\,\text{s}$, the overall time to replay all patches would amount to approximately $61\,\text{h}$ which was not acceptable in our particular agile use-case where we ran *SourceWarp* to inform the feature development process while building out the feature.

This is also the reason why manual testing was not an option for our use-case because manually cherry-picking relevant commits from S' alone, preparing the patch sequences, applying the patches and then manually documenting the resulting metrics/results would have been very time intensive (exceeding $61\,\text{h}$), error prone and not practical even more so in the presence of changing testing parameters such as time-frame, patch sampling rates, etc.

However, as many of the commits in S' are not relevant with respect to Vulnerability Tracking, we used the allow-list feature of *SourceWarp* to focus the attention on only relevant commits, i.e., commits that are touching those files that contain vulnerability findings. We extracted them from the Vulnerability Report that recorded all historical findings on the GitLab source project. With the allow-list we could reduce the slice size from $|S'| = 2748$ to $|S| = 131$ which significantly cut down the estimated average replay time by 95% to approximately $3\,\text{h}$.

Impact of the patch sampling number

We further reduced the replay time to $\sim 18\,\text{min}$ by using a patch sampling number of $N = 10$ so that only $|PS| = 13$ distinct patches were replayed during the monitor phase; a higher patch sampling number leads to more commits being grouped together, leading to a lower granularity and a lower

execution time. For our use-case $N = 10$ was an acceptable trade-off between granularity and replay time.

Concerning *RQ2*, the allow-list and patch sampling number enabled us to reduce the benchmarking and testing time from days to hours. This shows that these heuristics enable *SourceWarp* to scale to large, heterogeneous DevOps platforms.

V. Implementation

The *SourceWarp* approach described in this paper is implemented in a Ruby tool which is publicly available through the GitLab project https://gitlab.com/gitlab-org/vulnerability-research/foss/sourcewarp.

SourceWarp is implemented in Ruby and can be executed via the command line or as a Docker container where all the parameters such as allow-list or patch sampling number can be provided as command line parameters or environment variables.

You can find a detailed installation and configuration documentation on the GitLab project which also includes the evaluation scripts we used for the case study presented in this paper.

VI. Limitations

While the *SourceWarp* approach is conceptually agnostic with regards to the used SCM, the tool currently only supports Git[6] as it is the most widely used SCM [2]. In addition *SourceWarp* currently only natively works with the GitLab API and/or Docker. However, the tool is designed in such a way that support for other APIs and container technologies can be easily added.

At the moment *SourceWarp* only leverages the Git history; it does not leverage commit meta-data and does not perform any analysis on the content of the commits.

As mentioned in section III, *SourceWarp* works best for evaluating a single feature in isolation: if multiple heterogeneous chained CI/CD tools perform a task that is relevant with respect to the tested feature, they can be treated as a single system with a final output which we could then use for both testing and benchmarking. Hence, at the moment *SourceWarp* does not support running benchmarking and testing experiments for multiple features simultaneously.

VII. Verifiability and Threats to Validity

Verifiability: The *SourceWarp* tool and evaluation scripts are available on our tool website https://gitlab.com/gitlab-org/vulnerability-research/foss/sourcewarp.

With the parameters presented in the industrial case study and the documentation provided on the website, the results with regards to Vulnerability Tracking deduplication presented in this paper are reproducible. However we opted against publishing the allow-list since it may include security sensitive information with regards to source code vulnerabilities.

[6]https://git-scm.com/

Threats To Validity: Our evaluation is subject to threats to validity as any empirical study. The most important is threats to external validity: results from a finite sample are not necessarily generalizable towards the entire range of possibilities.

Our results are based on an industrial case study where we used *SourceWarp* to assess the impact of a newly introduced Vulnerability Tracking feature. The fact that we only conducted a single industrial case study in order to evaluate *SourceWarp* could be considered a threat to *external validity*.

However, the evaluation scripts are the only settings that are feature-specific so that in order to automatically collect data, the evaluation scripts have to have access to intermediate results/metrics as *SourceWarp* runs through the replay step. Apart from the evaluation scripts, the *SourceWarp* approach is agnostic with regards to the tested feature.

We would also like to note that for the industrial case study *SourceWarp* provided data that contributed to the successful product integration of Vulnerability Tracking by providing data for the purpose of DDDM. Vulnerability Tracking runs to this day in production.

VIII. Related Work

Test automation

Test automation approaches automate the task of creating a mechanically interpretable representation of manual test cases [7], [8], [9].

SourceWarp relates to test automation approaches as it synthesizes test input from a Git history which can be considered as test inputs for CI/CD tools. However, in contrast to the approaches above, *SourceWarp* is geared towards testing DevOps platforms and CI/CD tools with the main goal of performing DDDM in a fully automated manner. In addition to generating these test inputs, *SourceWarp* also replays them and assesses their impact.

Record & Replay Frameworks

Record & Replay approaches [10], [11], [12], [13], [14], [15] support recording and replaying system interactions in an automated manner. However, these approaches are usually focused on robustness/security testing with an emphasis on simulating user-interactions. In addition they often require a user or a crawler [16] to simulate interactions before they are able to replay them.

SourceWarp also uses record and replay phases. However, instead of recording user interactions, it systematically and automatically samples merge request commits from the SCM history that are relevant with respect to the tested feature before replaying them on the target SUT recording the results and evaluating the impact in the monitoring phase. To the best of our knowledge, *SourceWarp* is the first approach targeted towards testing heterogeneous DevOps platforms and CI/CD tools that is deeply integrated with the SCM in order to automatically source user interactions encoded in the code change history.

Mining Software Repositories

As *SourceWarp* performs a light analysis on data stored in SCMs in order to extract relevant commits and synthesizes commits based on that, it is conceptually related to Mining Software Repositories (MSR) approaches which focus on the analysis of the (meta-)data stored in SCMs. MSR approaches extract information from source code histories for test-case reduction [17], for machine learning applications [18], [19] or clone detection [20], etc.

While *SourceWarp*'s record phase also leverages the history of the SCM repository the approach is not geared towards collecting intelligence from the history of the SCM history. *SourceWarp* is first and foremost a testing tool that supports testing and benchmarking for the purpose of DDDM; all the data *SourceWarp* leverages from the SCM is selected to serve this purpose.

Variant Testing

A/B testing [5], [6], an instance of variant testing, helps with DDDM by running controlled experiments where not yet fully deployed, new features are evaluated by assessing interactions with a selected group of users presenting different variants of the test subject. A/B testing is most commonly applied to assess user interface changes. However, variant testing strategies such as A/B testing are usually applied to monitor customer experience, mostly through user-interface interactions. These forms of testing strategies are not always applicable to highly automatized environments.

Canary testing [3], [4] is a testing technique that reduces the risk of deploying a new feature by slowly transferring load from the current to a new *canary* version that ships the new feature.

In contrast to A/B testing, *SourceWarp* cannot be used to perform UI variant testing. However, as demonstrated in our case-study, *SourceWarp* can be used to perform differential analysis to assess the impact of a new feature *before* it is integrated in production which is not the case for A/B and canary testing that require the feature to be deployed at least partially. As opposed to A/B, which usually rely on human interaction, *SourceWarp* leverages the SCM as a foundation because it is focused on testing/benchmarking CI/CD tools and DevOps platforms where user interactions are already recorded in the SCM history so that *SourceWarp* can perform testing and benchmarking in a fully automated and reproducible manner.

Fault Localization

Delta Debugging [21] is a technique to isolate failure causes on user input which makes it very useful for isolating failure inducing changes on source code. While the initial approach was focused on monolithic systems, it has been extended and applied in other scenarios such as micro-services [22].

Other approaches focus on debugging services oriented applications by supporting the reproduction of failures by means of analysis tooling [23], by using model-based approaches to diagnose failures that occurred in the application [24] or by applying spectrum-based fault localization [25], [26] to pinpoint components that are most likely to contain faults.

SourceWarp is loosely related to fault localization approaches as the approach also tries to cherry pick certain code-changes that are most interesting with regards to the benchmarking exercise. Instead of localizing faults, *SourceWarp* implements an *allow-list* heuristic to cherry-pick commits related to files that are considered relevant. In addition *SourceWarp* is designed to work well in highly automated environments. However, *SourceWarp* is a benchmarking and testing approach and not geared towards debugging.

IX. Conclusion and Future Work

In this paper, we present *SourceWarp*, a scalable testing and benchmarking approach that is targeted towards DevOps platforms and CI/CD tools and supports both testing and benchmarking in a cost effective and reproducible manner. We present a real-world industrial case-study for which we successfully applied *SourceWarp* to test and benchmark a newly developed feature at GitLab which has been successfully integrated into the product. In the case study we demonstrate that *SourceWarp* scales towards large, industrial systems and is highly effective in supporting DDDM by providing automation for testing and benchmarking proof-of-concepts.

However, as *SourceWarp* can be basically used for all those cases where test input is provided by means of an SCM repository, we are planning to apply it for finding regressions in DevOps platforms and CI/CD tools that rely on structured input which is stored in a Git repository itself. For instance, the GitLab Security Advisory Database[7] as a file and Git-based security advisory database for 3rd party software dependencies stores over 16K files with a commit history of 38K commits; the database is used as input by different tools which we could test using *SourceWarp*.

Acknowledgements

This work would not have been possible without the tireless effort and support of multiple current and former team members especially from the Static Analysis Team[8] as well as the Threat Insights Team[9] at GitLab.

We thank Lucas Charles (*lcharles@gitlab.com*), Ross Fuhrman (*rfuhrman@gitlab.com*), Zachary Rice (*zrice@gitlab.com*), Taylor McCaslin (*tmccaslin@gitlab.com*), Connor Gilbert (*cgilbert@gitlab.com*), Thiago Figueiró (*tfigueiro@gitlab.com*), Hillary Benson (*hbenson@gitlab.com*), Thomas Woodham (*twoodham@gitlab.com*) and Wayne Haber (*whaber@gitlab.com*), Todd Stadelhofer, Saikat Sarkar and Daniel Searles.

We thank Dennis Appelt (*dappelt@gitlab.com*) for providing valuable feedback on earlier versions of the manuscript.

[7] https://gitlab.com/gitlab-org/security-products/gemnasium-db

[8] https://about.gitlab.com/handbook/engineering/development/sec/secure/static-analysis/

[9] https://about.gitlab.com/handbook/engineering/development/sec/govern/threat-insights/

REFERENCES

[1] KPMG, "Agile transformation," https://assets.kpmg/content/dam/kpmg/be/pdf/2019/11/agile-transformation.pdf, 11 2019.

[2] StackOverflow, "Stack overflow developer survey 2022," https://survey.stackoverflow.co/2022, 10 2022.

[3] A. Tarvo, P. F. Sweeney, N. Mitchell, V. Rajan, M. Arnold, and I. Baldini, "Canaryadvisor: A statistical-based tool for canary testing (demo)," in *International Symposium on Software Testing and Analysis (ISSTA'15)*. ACM, 2015.

[4] J. Humble and F. David, *Continuous Delivery*. Addision Wesley, 2010.

[5] S. Kamalbasha and M. J. A. Eugster, *Bayesian A/B Testing for Business Decisions*. Springer, 2021.

[6] P. L. Li, X. Chai, F. Campbell, J. Liao, N. Abburu, M. Kang, I. Niculescu, G. Brake, S. Patil, J. Dooley, and B. Paddock, "Evolving software to be ml-driven utilizing real-world a/b testing: Experiences, insights, challenges," in *International Conference on Software Engineering (ICSE'21)*. ACM/IEEE, 2021.

[7] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *International Conference on Software Engineering (ICSE'12)*. IEEE, 2012.

[8] T. A. Lau, C. Drews, and J. Nichols, "Interpreting written how-to instructions," in *Joint Conference on Artificial Intelligenceo (IJCAI'09)*. ACM, 2009.

[9] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan, "Koala: Capture, share, automate, personalize business processes on the web," in *SIGCHI Conference on Human Factors in Computing Systems (CHI'07)*. ACM, 2007.

[10] Z. Long, G. Wu, X. Chen, W. Chen, and J. Wei, "Webrr: self-replay enhanced robust record/replay for web application testing," in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*. ACM, 2020.

[11] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive record/replay for web application debugging," in *Symposium on User Interface Software and Technology (UIST'13)*. ACM, 2013.

[12] J. Thomé, A. Gorla, and A. Zeller, "Search-based security testing of web applications," in *International Workshop on Search-Based Software Testing (SBST'14)*. ACM, 2014.

[13] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra, "Robust test automation using contextual clues," in *International Symposium on Software Testing and Analysis (ISSTA'14)*. ACM, 2014.

[14] S. Thummalapenta, N. Singhania, P. Devaki, S. Sinha, S. Chandra, A. K. Das, and S. Mangipudi, "Efficiently scripting change-resilient tests," in *International Symposium on the Foundations of Software Engineering (FSE'12)*. ACM, 2012.

[15] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, "Coscripter: Automating & sharing how-to knowledge in the enterprise," in *SIGCHI Conference on Human Factors in Computing Systems (CHI'08)*. ACM, 2008.

[16] A. Mesbah, E. Bozdag, and A. van Deursen, "Crawling ajax by inferring user interface state changes," in *International Conference on Web Engineering (ICWE'08)*. IEEE, 2008.

[17] G. Gharachorlu and N. Sumner, "Leveraging models to reduce test cases in software repositories," in *International Conference on Mining Software Repositories (MSR)*. IEEE, 2021.

[18] G. Nguyen, M. J. Islam, R. Pan, and H. Rajan, "Manas: Mining software repositories to assist automl," in *International Conference on Software Engineering (ICSE'22)*. ACM, 2022.

[19] N. A. Nagy and R. Abdalkareem, "On the co-occurrence of refactoring of test and source code," in *International Conference on Mining Software Repositories (MSR'22)*. ACM, 2022.

[20] Q. Wu, H. Song, and P. Yang, "Real-world clone-detection in go," in *International Conference on Mining Software Repositories (MSR'22)*. ACM, 2022.

[21] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'99)*. Springer, 1999.

[22] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding, "Delta debugging microservice systems," in *International Conference on Automated Software Engineering (ASE'18)*. ACM, 2018.

[23] N. Arora, J. Bell, F. Ivancic, G. Kaiser, and B. Ray, "Replay without recording of production bugs for service oriented applications," in *International Conference on Automated Software Engineering (ASE'18)*. ACM/IEEE, 2018.

[24] M. Alodib and B. Bordbar, "A model-based approach to fault diagnosis in service oriented architectures," in *European Conference on Web Services (ECOWS'09)*. IEEE, 2009.

[25] C. Chen, "Automated fault localization for service-oriented software systems," Ph.D. dissertation, Dresden University of Technology, Germany, 2015.

[26] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," 2016.