Orca: Differential Bug Localization in Large-Scale Services

Ranjita Bhagwan Rahul Kumar Chandra Sekhar Maddila Adithya Abraham Philip

Microsoft Research India

Abstract

Today, we depend on numerous large-scale services for basic operations such as email. These services are complex and extremely dynamic as developers continuously commit code and introduce new features, fixes and, consequently, new bugs. Hundreds of commits may enter deployment simultaneously. Therefore one of the most time-critical, yet complex tasks towards mitigating service disruption is to localize the bug to the right commit.

This paper presents the concept of *differential bug localization* that uses a combination of differential code analysis and software provenance tracking to effectively pin-point buggy commits. We have built Orca, a customized code search-engine that implements differential bug localization. On-Call Engineers (OCEs) of Orion, a large enterprise email and collaboration service, use Orca to localize bugs to the appropriate buggy commits. Our evaluation shows that Orca correctly localizes 77% of bugs for which it has been used. We also show that it causes a 3x reduction in the work done by the OCE.

1 Introduction

Orion [1] is a large enterprise email and collaboration service that supports several millions of users, runs across hundreds of thousands of machines, and serves millions of requests per second. Thousands of developers contribute code to it at the rate of hundreds of commits per day. Dozens of new builds are deployed every week. Software bugs are bound to be common in such a complex and dynamic environment. It is critical to detect and promptly localize such bugs since service disruptions lead to customer dissatisfaction and significantly lower revenues [21].

When a service disruption happens because of a software bug, the first-step towards mitigating its effect is to localize the responsible bug to the right commit. We call this *commit-level bug localization*. This is a non-trivial task since the intense pace of development demands that multiple commits be aggregated into a single deployment. In addition, commit-level bug localization needs to happen as quickly as possible so that buggy commits can be reverted promptly thereby restoring service health. About half of all Orion's service disruptions are caused by software bugs.

Unfortunately, bug localization in large services such as Orion is a cumbersome, time-consuming, and error-prone task. The *On-Call Engineers (OCEs)* are the first points-of-contact when a disruption occurs, and they are responsible for bug localization. Though knowledgeable, on-call engineers can hardly be expected to have complete and in-depth understanding of all recent commits. Moreover, bugs that emerge after deployment are complex and often non-deterministic. And yet, very few tools exist to enable OCEs to perform this critical task.

Our goal is to build a tool that will help OCEs correctly and swiftly localize a bug to the buggy commit. Over a period of eight months, we studied post-deployment bugs, their symptoms, the buggy commits that caused them, and the current approach to bug localization that Orion's OCEs follow. We made four key observations.

1) *Bug localization is time-critical, bug fixing is not.* When a bug disrupts a service, the OCE's task is to keep the service disruption time to a minimum. She finds the buggy commit as fast as possible and reverts it rather than wait for the concerned developer to fix the bug. The reason is that, depending on the complexity of the bug, the developer may take a long time to fix it. Therefore to keep disruption to a minimum, it is better to revert the buggy commit first and introduce the fix at a later time. Thus, fast commit-level bug localization is critical.

2) *Rich monitoring infrastructure exists but is insufficient because of uncaptured dependencies.* Since service disruptions are a major concern, developers have created thousands of active *probes* that periodically monitor service-components or API calls and raise *alerts* if they fail. Despite this, bug localization is a challenge because a probe to a component may fail not because of any change to the component itself, but because of a change to another component that depends upon it. For instance, a server-side probe failed with an exception `Type RecipientId not supported` because a developer made a commit to client-side code that added support for the data type `RecipientId` without adding support on the server-side. To make matters worse, as the service evolves fast, new dependencies emerge at a rapid

[1] Name changed.

rate and no tool can completely capture all of them.

3) *Symptom descriptions and their causes tend to have similar terms.* We have found that when a probe detects a bug, a similarity often exists between terms in the unhealthy probe name or exception text that it generates and the source-code change that caused the bug. In the example mentioned in the previous paragraph, the term `recipient` occurs in both symptom (the exception text) and cause (added support to the data type on the client). We also see this similarity in some customer complaints as well which predominantly use natural language. For instance, a customer recently complained that "*Email ID suggestions for people I know is not working.*". The cause for this was an incorrect change to a function named `PeopleSuggest`.

4) *Bugs may appear well after the buggy commit is deployed.* We observed that while the symptoms of a bug appear in a current build, the cause may be a commit deployed in a much older build. These are particularly challenging for the OCE to localize because they have to investigate, in the worst-case, all commits in the current build before moving on to investigate an older build. This can significantly lengthen service disruptions.

Keeping these observations in mind, we design a novel search technique that we call *differential bug localization*. Using descriptions of the bug as a query, we detect changes to the abstract syntax tree in the source-code and search only these changes for text-based similarity. We call this *differential code analysis*. To find offending commits in older builds, we introduce a construct called the *build provenance graph* that captures dependencies between builds. We designed Orca, a custom code search-engine that leverages differential bug localization to provide a ranked list of "suspect" commits.

The Orion service has integrated Orca into its alerting and monitoring processes. This paper describes Orca and makes the following contributions:

- We provide a study of post-deployment bugs found in the Orion service and their characteristics (Section 3).
- We introduce *differential bug localization*, which uses two constructs: differential code analysis of the abstract syntax tree, and the build provenance graph. In addition, we use a prediction of commit risk to call out riskier commits in the list of potential root-causes (Section 4).
- We have designed Orca, a tool that Orion's OCEs are actively using to localize bugs (Section 5).
- We provide an evaluation of Orca for bugs found in the Orion service (Section 6).

To the best of our knowledge, ours is the first study of a bug localization tool deployed on a large-scale enterprise service. We have evaluated Orca on 48 post-deployment bugs found in Orion since October 2017. We show that Orca correctly localizes 37 out of 48 bugs for a recall of 77%. In 30 of the 37 cases, the correct commit was ranked in the top 5 records shown by our UI (Section 6). We also show that Orca causes a $3\times$ reduction in the work done by the OCE.

We have designed Orca for usability and ease-of-adoption. While this paper concentrates on Orion's deployment of the tool, Orca has been deployed on five other services within our enterprise. Our techniques are generic and extend well to other large services. To make it easy for OCEs to use the tool interactively, we have optimized its performance through multiple caching and preprocessing techniques. Our user-interface provides results with an average run-time of 5.9 seconds per query.

2 Related Work

The Programming Languages, Software Engineering and Systems communities have extensively studied bug detection, bug localization and debugging. While Orca takes inspiration from some of this prior work, it targets a fundamentally different application space, i.e. large-scale service deployments. Also, Orca is meant to be used by on-call engineers, not developers.

A bug localization tool for such services needs to be *fast*: the query response time should be at most a few seconds since OCEs will use the UI interactively. It should be *general*: the techniques should support code in different languages and should not need an OCE or developer to provide specification. It should be *non-intrusive*: we should not require any changes to the service's existing coding and deployment practices. Finally, it should be *adaptive:* it should work in an extremely dynamic and changing environment. We now describe prior work in the area and explain why it does not satisfy some or all of these requirements.

IR-based Bug localization techniques [3, 25, 35, 36, 38, 42, 49] use a given bug-report to search, based on textual similarity, for similar bug reports in the past. For each match, they localize the bug at the *file-level*, not at commit-level, to files that have been changed to address similar bug-reports. Wang et al. [38] present a structured and detailed study of the various techniques that are used for information-retrieval based bug localization. They use similar search techniques over five major concepts: version history, bug reports, stack traces, and reporter information.

While these techniques are fast, general and non-intrusive, they assume a stationary system, i.e. they as-

sume that there is an inherent similarity between the current version of the system and previous versions. This is fundamentally not true in service deployments. For instance, Orion experiences a change in module dependencies at the rate of 1% to 3% every month, and some source-code files consistently change more than once a day.

Moreover, prior work has mostly studied software products, not deployed services. Comparatively, the work presented in this paper is different because it focuses on (a) dynamic deployments of software services, and (b) both structured and unstructured queries for localization of the manifested issue. Finally Orca is deployed and being used in real-time on a large service deployment. To our knowledge, existing bug localization techniques have not seen such scale of deployment.

Dynamic Analysis techniques are the most commonly used and widely studied approaches for detecting bugs and issues in software. Testing and automated fuzz testing techniques [6, 18] provide an effective method to automatically generate test-cases that produce random inputs for the underlying software. Although these techniques are useful, they are complementary to our approach. Testing is never complete and does not provide guarantees about the correctness of the software. As a result, in spite of comprehensive testing, bugs still emerge regularly in service deployments, as we have noticed with Orion.

To find post-deployment bugs, previous work has presented statistical techniques [7, 9, 26] to automatically isolate bugs in programs by tracking and analyzing *successful* and *failed* runs of the program. While these technique hold promise, they are intrusive: they requires fine-grained instrumentation and a large number of program traces from the service deployment. Given the stringent performance needs and dynamism in services, we do not have the luxury of utilizing such techniques.

Delta Debugging techniques [12, 45] help automate the problem of debugging by providing the debugger or programmer information about the state of the program in passing and failing runs. The possible search space of the root cause is systematically pruned by using information from the various runs and by creating new executions. The ideas in delta debugging rely heavily on *program slicing* [1, 39]. Given its requirements, delta debugging tends to require a large number of tests and the data from instrumented programs. Hence, it is intrusive.

The GIT `bisect` command [17] is similar in flavor. Given the last good commit, it uses an algorithm based on binary search to go through subsequent commits, repeatedly test the code, and ultimately localize the problem to the buggy commit. However, like delta debugging, this may require a large number of testing cycles. Moreover, a bug in a large deployment may not be reproducible by simple testing.

Static analysis entails analyzing software without executing the programs in question. The analysis may be performed at the level of the source-code itself, or at the level of the object code that is generated for execution (byte code or binary). Analysis is performed by automated tools that tend to be rooted in some formal method such as *model checking* [11], *symbolic execution* [29], and *abstract interpretation* [13]. Although such techniques have shown significant promise in the past, performing such analysis on a large scale for services has proven to be very slow and intractable. Performing program analysis and verification at smaller scales for individual components is the current limit of such techniques.

Tools such as Semmle [14] provides a unified framework that implements various program analysis techniques and correctness checks. In our experience, Semmle has proven to be somewhat useful for simple situations, but it lacks generality.

Differential static analysis techniques such as SymDiff [24] are immediately relevant to the problem discussed in this paper. But differential analysis techniques are usually *property driven*; two versions of the program are analyzed with respect to a specific correctness property. For example, the analysis may be performed for asserting differences in the new version w.r.t. null pointer dereferences. We believe this approach too lacks generality since it is not feasible to enumerate all such properties of code in large dynamic services with multiple dependent components. Yet, we do draw inspiration from this work to build Orca's AST-based differential analysis.

Tools such as Gumtree [15] use differences in ASTs to derive accurate edit scripts. Their techniques need to be fine-grained and therefore use heavyweight algorithms that implement isomorphism detection. Orca uses differences in ASTs with a different goal: for a changed file, we use the AST difference to enumerate all changed entities, such as changed variables, methods, classes, namespaces, etc. So we use faster, more coarse-grained heuristics than Gumtree, making our techniques much more performant.

Log Enhancement techniques [44, 47] improve log-based debugging by making logs richer and more targeted towards diagnosability. We believe such work is complementary to our approach and Orca can gain significantly with such techniques.

3 Overview

In this Section, we first describe the system development lifecycle of the Orion enterprise email service and the role of the OCE. We next describe the characteristics

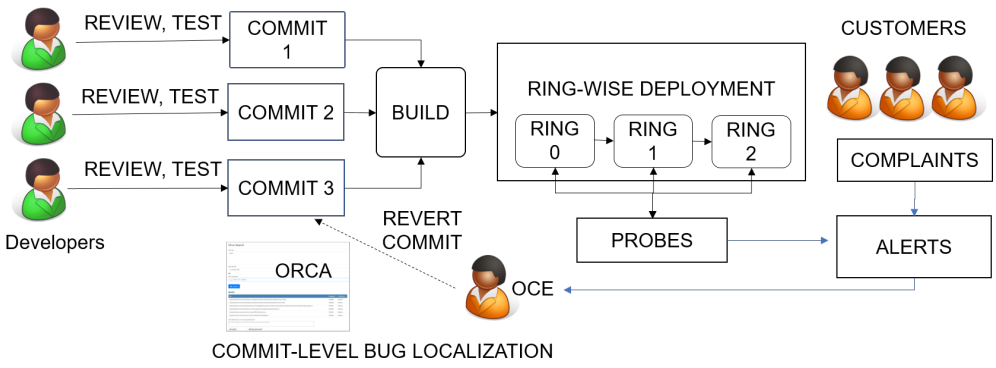

Figure 1: System Development Lifecycle (SDLC) of the Orion Service.

of post-deployment bugs in Orion and motivate the approaches we adopt in Orca. We provide an overview of Orca and its goals, and finally state Orca's scope.

3.1 System Development Lifecycle

Figure 1 shows Orion's system development lifecycle (SDLC) and Table 1 holds a summary definition of each term we use for the reader's convenience. Multiple developers *commit* code, where a commit varies in complexity from a small tweak to a single file to changes to hundreds, even thousands of files. These commits are *reviewed* by one or more reviewers that the developer chooses. After multiple iterations with the reviewers, the commits are tested using unit, integration, and functional tests. Periodically, the administrator creates a new *build* by combining a set of commits. A build is a unit of deployment of the service and may contain just one, or hundreds of commits.

Builds are deployed in stages onto *rings*. A ring is a pre-determined set of machines on which the same build runs. The build is first deployed onto the smallest ring, or "Ring 0", consisting of a few thousand machines. When it is considered safe, the build progresses through multiple rings such as Ring 1 and Ring 2 until it is finally deployed world-wide. The idea of this staged deployment is to find bugs early in the life-cycle.

Once the build is deployed to a ring, several tools monitor it. A tool may use passive or active monitoring techniques, either analyzing logs or sending periodic *probes* to a component. It uses anomaly detection techniques to raise an *alert* that the OCE receives.

If only the machines running a specific build raise alerts, the OCE concludes that the build is buggy and she begins bug localization. Roughly half of all alerts in Orion are caused by bugs. Consequently, bugs are a significant reason for service disruption. If the alerts are not confined to a particular build, the bug is likely due to other reasons such as faulty networks or hardware misconfiguration. Root-causing infrastructure issues is not our focus as several tools already exist for this purpose [43, 46].

To localize the bug to a commit, the OCE picks the commit that she feels is most likely to cause an issue and contacts the developer who created it. If the developer responds in the affirmative that her commit may indeed have caused the bug, the commit is immediately reverted and the service is restored to a healthy state. Note that the developer does not necessarily debug or fix the bug before responding to the OCE. If the developer says that their commit is not responsible for the bug, then the OCE picks the next most likely commit, and repeats the process until the service becomes healthy. Compared to a novice, an experienced OCE with domain-knowledge may pick the correct commits more promptly and therefore restore the service much faster. Orca removes this dependency on experience and domain-knowledge by codifying it in its search algorithms.

3.2 Post-Deployment Bugs

Over a period of eight months, we analyzed various post-deployment bugs and the buggy source-code that caused them. Table 2 outlines a few characteristic issues. The table shows the type of alert, it provides an overview of the symptom, and a description of the root-cause. It also shows the number of commits (and the number of files) that an OCE has to consider while performing bug attribution which, in some cases is more than 200.

In general, we have found that bugs fall predominantly into one of the following categories:

- **Bugs specific to certain environments.** Often, a component starts failing because files implementing that component have a bug (Bugs 2, 5). Usually, the failures happen only for a specific type of client such as web-based clients, or in a specific region such as Japan. Tests do not catch the bug since

Term	Definition
Commit	Set of file changes made by one developer.
Review	Recommendations made by one or more developers for a commit.
Build	Unit of deployment for the service consisting of one or more commits.
Ring	Set of machines onto which a build is deployed.
Probe	Periodic checks to functions/APIs to ensure they are working as expected.
Alert	An email- or web-based notification that warns the OCE of a problem.

Table 1: Terms used in the SDLC description and their definitions

not all configurations, clients and environments are tested.

- **Bugs due to uncaptured dependencies.** Dependencies can be of various types. In Bug 10, a server-side implementation is modified without appropriately modifying the client-side code. This happens because developers often overlook such dependencies as no compile-time tool captures them completely. Another example of an uncaptured dependency is Bug 4. A commit modifies a certain library, but unbeknownst to the developer, another component depends on certain features in the older version of the library and stops working correctly.

- **Bugs that introduce performance overheads.** Several probes track performance issues. For instance, in Bug 8, a code addition that was not thread-safe caused CPU overload that slowed down the service. The bug emerges only when a large number of users use the service. Hence it is not caught in testing.

- **Bugs in the user-interface.** A UI Feature starts misbehaving, so a customer complains. An example of this is Bug 1.

3.3 Orca Overview

Studying these bugs and observing the OCEs gave us valuable insights. We state these insights, and describe how Orca 's design is influenced by them.

Often, the same meaningful terms occur both in the symptom and the cause. Table 2 captures this under the "Term Similarity" column. Some matched terms are proper nouns such as the component name (Bug 3) or global data types (Bug 7). They can also be commonly used terms such as protocols (`Imap` in Bug 9) or the function performed (`suggest` in Bug 1 and `migrat` in Bug 6). Given the term similarity between symptom and cause, we designed Orca as a custom-designed search engine, using the symptom as the query-text, and giving us a ranked list of commits as results. Orca searches for the symptomatic terms in names of the modified code by performing differential code analysis on the abstract syntax tree. We describe this procedure in Section 4.1.

Testing and anomaly detection algorithms do not always find a bug immediately. A bug may start surfacing in a new build despite being introduced through a commit to a much older build. Bug 3 in Table 2 is an example. We introduce a *build provenance graph* to allow Orca to expand its search to older builds from which the current build has been derived. We describe this in Section 4.2.

Builds may have hundreds of commits, so manually attributing bugs can be long-drawn task. For instance, Bug 2 appears in a build that had 201 commits. Bug 3 appears in a build with 160 commits but the root-cause was in the previous build which had 41 commits. The OCE is faced with the uphill task of analyzing, in these cases, up to 200 commits before discovering the buggy commit. OCEs often work at odd hours and are constantly pressed for time. Orca therefore ranks commits based on a *prediction of commit risk.* Orca uses machine-learning and several features such as developer experience, code hot-spots and commit type to make this prediction. We describe this in Step 4 of Section 4.3.

To facilitate its use, we have built an Orca user interface and leverage caching and parallelism to ensure an interactive experience for the OCE. We describe our optimizations in Section 5.

There are thousands of probes in the system, and probe failures and exceptions are continuously logged. Therefore there is rich data on what symptoms, or potential queries to Orca look like. This allows us to track frequency of terms that appear in the queries and use the *Inverse Query Frequency (IQF)* rather than the Inverse Document Frequency (IDF) in our search rankings. We explain this further in in Section 4.3.

3.4 Orca Scope

In this section, we elaborate what Orca does *not* aim to solve. This is primarily because existing techniques already address these issues.

Orca does not target issues caused by faults in the infrastructure. Several techniques [22, 43] exist to do this. Orca does not solve the anomaly detection problem directly as several techniques already exist for this [4, 10].

No.	Type	Symptom	Cause	Term Similarity	Commits
1	Customer complaint	"People Suggestion" feature, that suggests potential recipients for an email, was not working for a subset of users.	A "people ranking" algorithm was incorrectly modified.	A variable used the keyword `suggest` in the modified function.	33
2	Probe	An email synchronization problem was detected.	A buggy commit to the synchronization component caused requests coming only from web-based clients to fail.	The probe contained the name of the synchronization component, which was also in the directory path of the modified files.	201
3	Probe	A worker process for a specific component started crashing repeatedly.	Incorrect configuration changes to the component's environment caused this. The commit was to a previous build but bug showed later only after a large number of users hit it.	The component name matched in the class name of modified code.	201
4	Customer complaint	Authentication process for some applications that used REST started crashing.	A library that these applications depended upon was modified but was not tested for all applications.	Keyword `auth` was in the pathname of the change.	46
5	Probe	Threads were getting blocked in processing, large delays were noticed in REST calls made by a web service.	HTTP client code for the service had been modified to make some synchronous calls asynchronous.	The component name matched a modified user-agent string in the HTTP client.	18
6	Probe	No. of exceptions generated anomalously high while migrating mailboxes	Caused by a code-change to a mailbox migration components.	Keyword `migrat` matched a changed function's name.	12
7	Probe	Number of exceptions in the log file for a component **C** became abnormally high.	Support for a new data type was added in a component that made an API call to component **C**, but **C** does not support that data type.	The exception text for **C** contained the data type.	70
8	Probe	CPU Usage on a set of machines was anomalously high.	Reads and writes to a dictionary were not thread-safe. Multiple threads were reading from a dictionary while it was being modified, causing a CPU blowout.	No keyword matched.	89
9	Probe	POP and IMAP services started failing.	Dependencies were broken when a code commit changed a library that the POP and IMAP services used.	Keywords `Pop` and `Imap` matched in code changes.	110
10	Customer complaint	A client signing in via OAuth does not display calendar.	Client-side implementation was incompatible with the server-side commit.	Keyword `OAuth` matched the symptom and the server-side change.	39

Table 2: Examples of post-deployment bugs.

But we do recognize that anomaly detection algorithms are imperfect. Orca's build provenance graph helps find bugs even when the anomaly detection algorithm detects a bug well after a commit introduces the bug.

Orca does not handle bugs where the query does not have any context-specific information. Notable examples are performance issues, where the symptoms are, simply, out-of-memory exceptions or CPU-above-threshold exceptions. There exist other techniques [10, 48] in the literature which can debug such issues and therefore, can be used in combination with Orca. We discuss this further in Section 7.

4 Design

In this Section, we describe Orca's differential bug localization constructs in more detail. The input query to Orca is a symptom of the bug. This could be the name of the anomalous probe, an exception message, a stack-trace, or the words of a customer-complaint. The idea is to search for this query through *changes* in code or configurations that could have caused this bug. Thus the "documents" that the tool searches are properties that changed with a commit, such as names of files that were added, removed or modified, commit and review comments, modified code and modified configuration parameters. Orca's output is a ranked list of commits, with the most likely buggy commit displayed first.

First, we describe two novel constructs that we use for search-space pruning and search-space expansion respectively: differential code analysis, and the build provenance graph. Next, we describe the machine-learning based model of commit risk prediction which Orca uses to determine a rank-order of probably root-cause commits. Finally we provide a detailed description of the search algorithm.

4.1 Differential Code Analysis

Orca can search the entire code file to perform bug localization. But this approach can find false matches and drop Orca's precision, especially since we have found that every commit changes, on average, only about 20 lines of source-code per-file (including file additions), whereas the entire file can consist of hundreds of lines of code.

Another simple approach, on the other end of the spectrum, would be to search only the file names for matches, and not look into the code at all. While this is partially effective, our evaluation in Section 6 shows that this does not give us satisfactory recall.

We therefore employ a middle-path – differential code analysis – within Orca. Prior work has studied differential code analysis, albeit mostly at the semantic level [23]. It identifies relevant pieces of the code change that can potentially cause different behavior in the new version relative to the old version, but with reference to a specified *property*, such as null dereference, memory consistency, etc. Such techniques rely on compuationally expensive techniques such as differential symbolic execution [34] and regression verification [19].

We do not go with the semantic approach because (a) it is difficult to determine the full set of properties to capture all bugs in large-scale services, and (b) we would like to avoid the performance overhead of traditional static analyzers for differential analysis.

On the other hand, we could go with a complete *lexical* analysis on the differences, i.e., we can match terms in the query with terms in the difference, without any syntactic or semantic understanding. This approach will be relatively lightweight. However, it will miss several root-causes because very often, terms that match, such as protocol names, are parts of higher-level structures such as the method names and classes that have been modified. Consider Bug 6 in Table 2, for instance. the term `migrat` appears in the name of the function that has been changed, but not in text that has changed. Our techniques therefore need to identify *syntactic* constructs, such as methods and classes, that have changed.

Therefore, rather than going with a semantic or lexical analysis, we perform a syntactic analysis. We use the abstract syntax trees (AST) of the old and new version of the program to discover relevant parts of the source that have been *added*, *removed*, and *modified*.

Our analysis finds differences in entities of the following types: class, method, reference, condition, and loop. We create a "difference set" D of two ASTs, A_{old} and A_{new}, in the following way. Say e_i is the old version of an entity, and e_j the new version. Say t is the type of the entity. Then,

$$D = \begin{cases} d_{added} = \forall e \in A_{new} \mid e \notin A_{old} \\ d_{removed} = \forall e \in A_{old} \mid e \notin A_{new} \\ d_{changed} = \forall e_{new} \in A_{new} \mid type(e_{new}) = t \\ \qquad\qquad \wedge e_{new} \in A_{old} \wedge e_{new} \neq e_{old} \\ \qquad\qquad \wedge type(e_{old}) = t \\ d_{diff} = e_{old} \Delta e_{new} \mid e_{new} \in A_{new} \\ \qquad\qquad \wedge type(e_{new}) = t \wedge e_{old} \in A_{old} \\ \qquad\qquad \wedge e_{new} \neq e_{old} \end{cases}$$

Thus, the difference set D captures entities that have been *added*, *removed*, or *changed*. For all entities in D that have been changed, we also capture the differences (d_{diff}) between the two versions of the entity using a heuristic. For instance, say our heuristic detects that two lines of a function F have been changed. In addition to D containing the name of the function F, D also includes d_{diff}, which contains the entire text of the two changed lines: both the old version, and the new version.

We would like to point out that our syntactic approach to differential code analysis is not sound: we may detect changes even though there are none. For instance, consider the case where a function name changes completely, but the body remains unchanged. Our algorithm will treat this as a completely new function. While this could cause a precision drop in Orca since we are including more textual differences than actually exist, the algorithm does ensure that all changes are captured.

4.2 Build Provenance Graph

In Section 3.2, we have shown that a buggy commit to an older build may show symptoms only in a subsequent build. This could be because an inaccurate anomaly detection algorithm detects an anomaly too late. It may also be that a subtle bug manifests only in certain environments or only when a large number of users hit the service.

To accommodate this scenario, we expand our search to include previous builds that the symptomatic build is "derived" from. We maintain a *build provenance graph (BPG)* that captures dependencies between various builds along the axes of time and ring ID. Figure 2 shows an example fragment of a build provenance graph.

4.2.1 Construction

We now describe how we construct the build provenance graph. The BPG captures how builds are created and promoted in the different rings. Every build is represented as $B^i_{r,v}$, where i is a the build identifier, r is the ring identifier, and v is the version of the build within a ring.

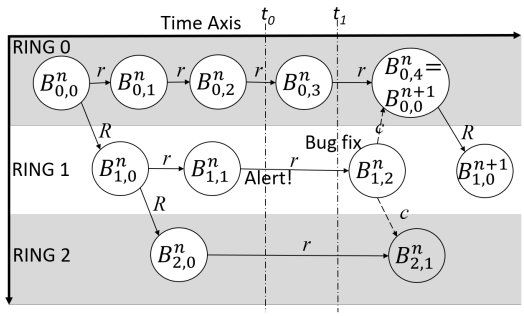

Figure 2: Example fragment of a build provenance graph.

Say a build spends a certain amount of time in Ring 0. If it is stable and shows no unhealthy behavior for a while, it is considered for promotion to the next ring. Build $B^n_{0,0}$ is one such build. It forks off build $B^n_{1,0}$ which runs in the next ring, Ring 1. Since $B^n_{1,0}$ is directly derived from $B^n_{0,0}$, any bugs that emerge in $B^n_{1,0}$ could potentially be due to commits originally made to $B^n_{0,0}$. Thus, we introduce an *inter-ring* edge between the two builds. Similarly, once a build is considered stable in Ring 1, it is forked off to Ring 2. This introduces the inter-ring edge between $B^n_{1,0}$ and $B^n_{2,0}$. For a given build identifier n, only one inter-ring edge can exist between two consecutive rings.

Meanwhile, developers make fresh commits $within$ each ring too, thereby creating the next build versions within the same ring. So in Figure 2, $B^n_{0,1}$ is derived from $B^n_{0,0}$, $B^n_{1,1}$ from $B^n_{1,0}$, and so on. We call the edges between these builds *intra-ring* edges. Several such intra-ring edges may exist in all rings though most are in Ring 0 which is the most dynamic and experimental ring.

At any given time, a vertical line drawn through the graph yields the *active build* within each ring at that time. For instance, at time t_1, the active builds are $B^n_{0,3}$, $B^n_{1,1}$, and $B^n_{2,0}$. For ease of explanation, we assume that at any given time there is only one active build in every ring though in reality there could be many.

We now explain a third edge-type called the *back-port* edge, shown in Figure 2 by dotted arrows. A critical bug that goes undetected may, with time, propagate to a large number of builds across all rings. Say at time t_0, such a bug causes an alert in build $B^n_{1,1}$. Say at time t_1, using the process described in Section 4.2.2, we localize the bug to a commit made to the earlier build $B^n_{0,0}$. The bug is fixed through a new commit c to the code of $B^n_{1,1}$, and this generates $B^n_{1,2}$, with an intra-ring edge between them.

We can see that since the bug originated in $B^n_{0,0}$, it also exists in the active builds within Ring 0 and Ring 2 that have been derived from it, i.e. $B^n_{0,3}$ and $B^n_{2,0}$. Consequently we apply the commit c, or we *back-port* it, to $B^n_{0,3}$ and $B^n_{2,0}$. This creates new builds $B^n_{0,4}$ in Ring 0 and $B^n_{2,1}$ in Ring 2. Thus we add two back-port edges with the label c from $B^n_{1,2}$ to $B^n_{0,4}$, and $B^n_{1,2}$ to $B^n_{2,1}$.

Finally, we describe how the build identifier n gets incremented in the build provenance graph. All builds across all rings that have the build identifier n are derived originally from $B^n_{0,0}$. Thus $B^n_{0,0}$ is called an *origin* build. With time, several new commits are applied in Ring 0. To ensure that these commits are fully deployed across all rings, a subsequent build from Ring 0 is chosen to be the next origin build. In the figure, this is $B^n_{0,4}$, which we rename as $B^{n+1}_{0,0}$, or the *next* origin build. All subsequent builds are now derived from this new origin build.

It can be seen that, barring backport edges, every node in the build provenance graph has only one incoming edge. This can be either an inter-ring or an intra-ring edge.

4.2.2 Traversal

We now describe how Orca uses the build provenance graph to expand its search-space and find potential buggy commits in older builds. Given a symptomatic build $B^i_{p,q}$, the purpose of the traversal is to find a list of candidate commits for the search.

We observe that Ring 0 is the most experimental of all rings. Builds in Ring 0 see a large number of significant commits. Consequently, our intuition is that, to localize a bug that appears in $B^i_{p,q}$, we should search all builds back to the the origin build $B^i_{0,0}$, which is in Ring 0. Thus using inter-ring and intra-ring edges, we backtrack from $B^i_{p,q}$ to the origin build $B^i_{0,0}$. In addition, we also include all back-ported commits to every build on the same path. Since every build has only one incoming inter-ring or intra-ring edge, there is only one such path from $B^i_{p,q}$ to $B^i_{0,0}$. The candidate list of commits to search will include all commits made to the builds on this path, and the back-ported commits on the same path.

We now explain this through an example with Figure 2. Say an alert is raised in $B^n_{2,1}$. Backtracking from $B^n_{2,1}$ to $B^n_{0,0}$ yields the set of commits $\{C(B^n_{2,1}), C(B^n_{2,0}), C(B^n_{1,0}), C(B^n_{0,0}\})$, where $C(B_{i,j})$ is the set of commits that were made to build $B_{i,j}$. To this, we add c, which is a backported commit from $B^n_{1,2}$ to $B^n_{2,1}$, thereby giving us the final list of commits to search in. That is,

$$\Gamma = \{C(B^n_{2,1}), C(B^n_{2,0}), C(B^n_{1,0}), C(B^n_{0,0}), c\}$$

4.3 Algorithm

In this Section, we describe Orca's search algorithm which uses differential code analysis and the build provenance graph. The Orca search algorithm consists of four

steps: 1) Query pre-processing where we perform tokenization, stemming and stop-word removal, 2) build graph traversal described in Section 4.2.2, 3) token-matching in code changes using Differential Code Analysis and 4) ranking and visualization of results. Our system runs differential code analysis and constructs the build provenance graph in the background periodically so that these tasks do not slow down the query response time.

Step 1: The search queries are symptoms of the problem, consisting of probe names, exception texts, log messages, etc. We first tokenize the terms in these symptoms by using a custom-built code tokenizer. This tokenizer uses heuristics that we have built specifically for code and log messages, such as splitting large complex strings along Camel-cased or Pascal-cased fragments. We also create n-gram based tokens since we have found that bigrams, such as `ImapTransfer` and `mailboxSync`, capture important information.

Next, we filter out irrelevant words also called stop-words [27] from these symptoms. Previous work has shown that logs have a lot of inherent structure [40]. For instance, all exception names have the suffix `Exception` and almost all log messages have a timestamp. Unlike conventional search-engines, even before we built Orca, we had access to about 8 million alerts consisting of probe names, exceptions and log messages from Orion's log store. We therefore perform stop-word removal on these to weed out commonly used or irrelevant terms such as `Exception` or timestamps. This step gives us a list of relevant "tokens" in the symptom. For each token t, we also maintain an *Inverse Query Frequency (IQF)* value [41] that we call t_{IQF}, obtained by analyzing Orion 's logs. t_{IQF} is calculated as (No. of queries/No. of queries in which token t appears). A high value of t_{IQF} implies that the token t is more important.

Step 2: We traverse the build provenance graph to find all builds related to the symptomatic build. From each build we discover, we enumerate all the commits that created the build. This leads us to the next step, which is matching tokens to files for each commit.

Step 3: Within a given commit C, for each file f and token t in the symptom, i.e for each tuple $T =< f, t >$, we search for the token in the difference set of the file, D_f. We use *TF-IQF* [41] as a "relevance" score, R_T^C, for each tuple. R_T^C is calculated as $n * t_{IQF}$ where n is the number of times the token t appears in difference set. This relevance score captures that the tuple $< f, t >$ is more relevant if the token t is very infrequent (i.e. t_{IQF} is very high), or if it appears many times in f.

We repeat this step for every token and file in the commit. At this point, we have file-level relevance values. Note though that we perform commit-level bug localization. Thus, we now aggregate the relevance values across all files and tokens to get one relevance value for the commit C, that we call R^C. So,

$$R^C = \underset{T}{\Theta} R_T^C \tag{1}$$

where Θ is an aggregation function such as such as `Max`, `Avg` or `Sum`. We show in Section 6 that the MAX function provides the best results in our deployment.

Step 4: Finally, Orca returns a list of commits in decreasing-order of their relevance. However, in deployment, we found that ranking solely based on decreasing order of relevance was not enough. Quite often, more than one commit had the same relevance score because several commits made at similar times matched the search terms equally.

We therefore build a machine-learning model that predicts commit risk to break the tie between commits that have the same relevance. This model uses ideas from a vast body of prior-work in this space in the Software Engineering community [5,20,30,32]. However, we believe ours is the first tool to apply such a risk prediction model to bug localization at the commit-level.

We have built a regression tree-based model that, given a commit, outputs a risk value for it which falls between 0 and 1. This is based on data we have collected for around 93,000 commits made over 2 years. Commits that caused bugs in deployment are labeled "risky" while those that did not, we labeled "safe". We have put in considerable effort into engineering the features for this task. The features that we input to the learner roughly fall into four categories:

- **Developer Experience**. Developers who are new to the organization and to the code-base tend to create more post-deployment bugs. Hence, we associate several experience-related features with each commit.
- **Code ownership.** We found that certain files that were mostly changed by a single developer caused fewer bugs than files that were constantly touched by several developers. Hence, for each commit, we use features to capture whether it touched files with very few owners or many owners.
- **Code hotspots.** Certain code-paths, when touched, tend to cause more post-deployment bugs than others. Some of our features capture this fact.
- **Commit complexity.** Several features, such as file types changed, number of lines changed, and number of reviewer comments capture the complexity of the commit.

Thus, for commits that have the same relevance score based on the terms match, we use the commit risk as a

secondary sort key to obtain a rank-order for Orca's output.

4.3.1 Example

We shall use the following example to illustrate how the search algorithm works. Say commits C_1 and C_2 create a build B. Say a probe called `LdapAuthProbe`, that monitors the LDAP authentication service, starts throwing exceptions of type `AuthFailedException`. Let us also say that the bug was caused by commit C_2 that erroneously modified a function `LdapRequestHandler` in a class `LdapService`, declared and defined in a file `LdapService.cs`. Say C_1 modified an `Imap` protocol implementation in a file `Imap.cs`.

The query to Orca is "`LdapAuthProbe AuthFailedException`". First, we tokenize and stem the query, and remove stop-words `Failed` and `Exception`. This yields the tokens `Ldap`, `Auth` and `Probe`. The word `Probe` occurs very frequently across all symptoms and therefore receives a very low IQF score, whereas `Ldap`, being a specific protocol, gets the highest IDF score. `Auth`, being somewhat more frequent than `Ldap`, receives a slightly lower IQF score.

We leave out build graph traversal for the sake of simplicity. Therefore, our list of candidate commits are only C_1 and C_2. In our example, the token `Ldap` will match both the class name, `LdapService`, and the function modified, `LdapRequestHandler`. Therefore the value of relevance for this is $2*$ the IDF value of token `Ldap`. C_2 will also find a match with token `Auth`. C_1, however, does not match any of these tokens. Our ranking algorithm will therefore choose C_2 over C_1 and, as the highest-ranked result, it will show the filename `LdapService.cs` and token `Ldap` rather than `Auth`.

5 Implementation

We have implemented Orca in a combination of C# (using .NET Framework v4.5) and SQL. We use the FastTree [16] algorithm within the ML.Net [31] suite for our commit risk prediction. Currently, the implementation is approximately 50,000 lines of code. In this Section, we briefly describe our implementation of Orca, and the various user interfaces we expose for OCEs.

5.1 Data Loaders

Since Orca requires information about various different parts of the system – source-code, builds, deployment information and alerts – a significant part of our implementation are data loaders for these different types of data. Figure 3 shows an architectural overview of the implementation. At the heart of Orca is a standard SQL database. This database is populated by data loaders at a predefined frequency. We now describe the different data loaders we use.

Source Data We implemented loaders for various source-control systems such as GIT and others internal to our organization. These loaders ingest source-code, code-versions and histories. The differential code analysis algorithm uses data from this loader.

Builds Data about builds resides in multiple big-data logs. We have build loaders that interface with several big-data logging systems to load build-specific information into our SQL database. We use this loader to construct the build provenance graph.

Deployment and Machine Data We load logs created by continuously monitoring the state of all machines within all rings. The state includes information about the current status of the machine (healthy/unhealthy), along with the information on the build version running on the machine.

Alerts are loaded from existing databases. Today, Orca supports multiple data sources for loading alert information.

5.2 Background Analyses

As our data loaders periodically load new data into the SQL database, we periodically initiate differential code analysis, build provenance graph construction, IQF calculation, and the commit risk prediction used in Section 4.3. If needed, the frequency of an analysis can be changed to make it more/less frequent. For example, the IQF calculation runs once a week, while commit risk prediction runs once every day. The other two processes run once every hour. Finally, it should be noted that all analyses that have been developed and deployed within this system are agnostic of the data source. The SQL database schema is normalized; thus, providing the same interface to all analyzers irrespective of the data source.

5.3 API Implementation

We now describe the Orca API and its implementation. Each Orca request is processed in real-time by the core Orca engine, and results are returned in JSON format. Clients decide on the relevant parts of the return result and how to display them. We make use of a Redis Cache [8] for improving our lookup times associated with data that is static. This includes data about the

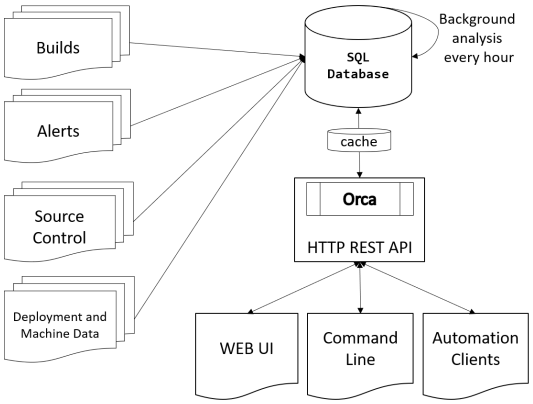

Figure 3: Implementation of Orca.

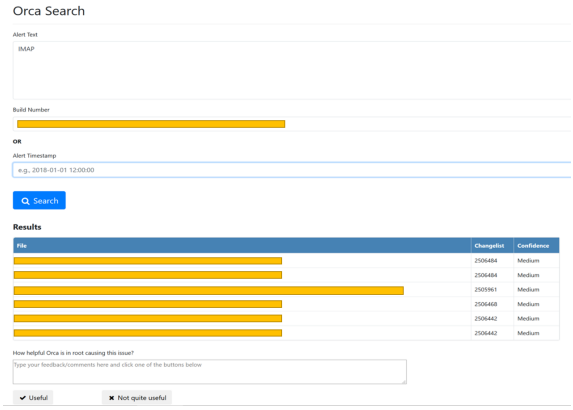

Figure 4: Web based UI of ORCA for Orion.

builds, files, source-code, difference sets, and the build provenance graph. Internally, the Orca system records all requests made and any feedback provided, which we use for learning and improvement.

All our services, servers, databases, and caches are implemented and operated using Azure as both a Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) provider.

5.4 Usage

Orca can be used in two ways: OCEs can use it to make *ad-hoc queries* interactively, or an alerting infrastructure can query Orca to get a list of suspect commits for an alert and include it in the alert itself.

For the OCEs, we have built a web-based UI that allows them to enter the details of their query and view the results. Figure 4 shows a screenshot of the UI with sensitive information removed. We have also built a PowerShell® *cmdlet* with which the OCE can interact with through a command-line interface.

To integrate Orca with alerting infrastructure, we have built an API that the alerting system can query directly. Currently, this mechanism is being used by multiple groups that are generating alerts within the Orion group. The web-based UI and *cmdlet* tend to be used by OCEs when new information such as log text or exception text has been discovered after the original alerts were generated.

Today, Orca has been deployed on multiple code-bases for six large-scale services within our enterprise. The combination of data loaders used in each code-base is slightly different and unique, but fully operational and functional.

6 Evaluation

In this section, we provide results we obtain by evaluating Orca. First, we evaluate Orca's result quality, i.e. how often it attributes a bug correctly to the right commit. Next, we evaluate how much effort an OCE saves by using Orca. Third, we evaluate the performance of Orca and the savings we get by using a Redis-based cache.

Since its deployment with Orion in October 2017, the Orca API has been invoked 4400 times to debug issues within the Orion service. Unfortunately, there is no central location where OCEs retrospectively log information about buggy commits. To evaluate how well Orca localizes a bug, we not only need the complete symptom of the bug, such as error messages or exceptions, but also the root-cause commit. Consequently, we begun a manual process towards gaining this information for as many bugs as we could. We interviewed multiple OCEs and manually analyzed source-code, bug-reports and email-threads.

By performing this exercise, we collected complete information for 48 of these bugs. These bugs vary greatly in characteristics. While some were inadvertently introduced by a single line-change, others were caused by complex dependencies between components.

6.1 Result Quality

To measure the quality of Orca's results, we interviewed several OCEs about how they would quantify result quality. Based on these interviews we determined that it is important that we find the buggy commit in as many cases as possible. This is captured by the *Recall*, i.e. the fraction of bugs where we found the buggy commit in any position in our results. The OCEs also told us that they linearly scan the list of commits that Orca provides, hence the closer the correct commit is to the top, the more time they save. To capture this, we use the *Mean Recip-*

	No. of results = 5			No. of results = 10			No. of results = 20		
Agg.fn.	DCA	DCA+BPG	ALL	DCA	DCA+BPG	ALL	DCA	DCA+BPG	ALL
MAX	0.65(31)	0.60(29)	0.63(30)	**0.69(33)**	**0.77(37)**	**0.77(37)**	0.69(33)	0.77(37)	0.77(37)
SUM	0.52(25)	0.52(25)	0.60(29)	0.71(34)	0.67(32)	0.77(37)	0.73(35)	0.79(38)	0.77(37)
AVG	0.60(29)	0.46(22)	0.58(28)	0.67(32)	0.73(35)	0.77(37)	0.69(33)	0.75(36)	0.77(37)

Table 3: We use recall to evaluate applying differential code analysis alone (DCA), differential code analysis with build provenance graph (DCA+BPG), and differential code analysis with build provenance graph and commit risk (ALL). Numbers in parentheses are the number of bugs correctly localized. We evaluate aggregating by MAX, SUM, and AVG.

	No. of results = 5			No. of results = 10			No. of results = 20		
Agg.fn.	DCA	DCA+BPG	ALL	DCA	DCA+BPG	ALL	DCA	DCA+BPG	ALL
MAX	0.41	0.38	0.39	**0.44**	**0.38**	**0.42**	0.44	0.38	0.42
SUM	0.42	0.36	0.40	0.44	0.38	0.43	0.44	0.39	0.43
AVG	0.36	0.25	0.38	0.37	0.29	0.41	0.37	0.29	0.41

Table 4: We use MRR to evaluate applying differential code analysis alone (DCA), differential code analysis with build provenance graph (DCA+BPG), and differential code analysis with build provenance graph and commit risk (ALL). We also evaluate aggregating by MAX, SUM, and AVG.

rocal Rank (MRR) [28]. MRR is the most suitable metric since we assume there is only one buggy commit that causes the symptom. MRR is calculated as $\frac{1}{n}\Sigma_{i=1}^{n} 1/r_i$, where n is the number of queries, r_i is the rank of the buggy commit for query i. If Orca is unable to find the correct commit, we assume r_i is infinity, i.e. we add 0 to the sum total.

We first evaluate differential code analysis (DCA) only. Next, we add the build provenance graph, without the commit risk-based ranking (DCA+BPG). Finally, we add commit risk prediction and evaluate Orca using all the techniques described in the paper (ALL). Tables 3 and 4 show the results for various combinations of parameters and features. We have varied the number of results we return as part of the Orca API to evaluate the quality for 5, 10 and 20 results, and we have evaluated Orca for different aggregation functions (Θ in Equation 1): MAX, SUM and AVG.

To evaluate Orca, we ask three questions:

- **How much value does differential code analysis add?** We wanted to understand whether looking into code was necessary.
- **How much value does the build provenance graph and commit risk prediction add?** We wanted to understand in what number of cases the build graph helped improve result quality. In addition, we wanted to see whether our rank-order based on commit-risk helped improve our results.
- **How should we aggregate the file-level relevances to commit-level?** Equation 1 in Section 4.3 described how we need to aggregate file-level relevance value into one value at the commit-level.
- **How many results should we show in the Orca UI?** When asked, the OCEs mentioned that they would not want to see beyond 10 results for each query. Hence, we wanted to evaluate the trade-off between the number of results shown and the recall and MRR.

We now answer each of these questions in order.

The build provenance graph adds 8% to the recall. Observe the data in bold in Table 3. DCA alone localizes 33 bugs for a recall of 0.69. Adding the build provenance graph helps us localize 4 more bugs correctly thereby increasing our recall to 0.77. While at first glance, this may appear to be a small increase, OCEs find it significantly more difficult to attribute bugs that occur in older builds, and therefore the value of finding these 4 bugs eases the OCE's workload considerably. We show this quantitatively in Section 6.2.

Adding commit risk-based ranking improves MRR by 11%. Observing the data in bold in Table 4, one can see that adding the build provenance graph reduced our MRR from 0.44 to 0.38. This happens because the build provenance graph increases our search-space significantly, and this increases the number of false-positive matches between terms and commits. Here, adding our secondary rank order based on commit risk improved our results by restoring the MRR value to 0.42.

The MAX and SUM aggregation functions per-

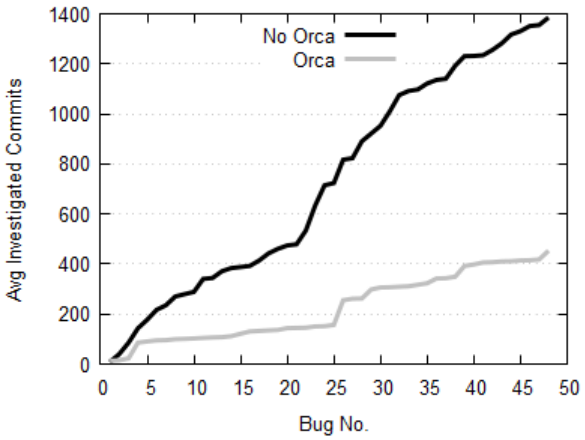

Figure 5: For all 48 bugs, a cumulative distribution function of the expected number of commits that the OCE investigates without Orca and with Orca.

form better than AVG. While the buggy code-changes match some very high-relevance tokens, several lower-relevance tokens match these code-changes too. Hence taking the average value across all matches dilutes the high-value token matches, therefore reducing both recall and MRR. Such a dilution does not happen if we use MAX or SUM. We choose MAX in our implementation.

Showing 10 results seems a good trade-off between result quality and UI succinctness. We evaluated results while setting the number of results shown as 5, 10 and 20. We find that with 10 results we achieve close to our best recall and MRR values.

With 10 results and using MAX, we obtained a recall of 0.77, i.e. we found the root-cause in 37 out of 48 cases. The MRR was 0.42. We also studied the matched terms for these 37 bugs and found that term-similarity serves as a good proxy to capture different types of bugs. Table 2 showed that matched terms fall roughly into four categories: they either match a component name, a function that the component performs such as `migrat` or `suggest`, data types, or protocol names such as `Imap`. We found in our case that of the 37 correctly localized cases, in 14 cases the token was a component name, in 17 cases it captured the function being performed, in 4 cases it matched a protocol name, and in the remaining 2 cases, the match was on a data type. Therefore term-similarity is quite versatile: it helps us catch a variety of bugs.

Of the 11 cases that we could not localize, in 1 case the issue was related to performance. In 4 cases, the problem was because a configuration setting. changed, and that triggered the use of code that was committed much earlier than our build provenance graph covered. In future work, we therefore plan to include all configuration settings in our differential analysis. In the remaining 6 cases, term similarity just did not capture the high-level semantics of the bug, and static or dynamic analysis may be required.

6.2 Reduction of OCE Workload

We now investigate the effect of Orca on reducing the OCE's workload. There are multiple ways to measure this. One way is to measure the amount of time saved by Orca for the OCE. Another is to determine the decrease in the number of commits that the OCE needs to manually investigate, both with and without Orca. We chose the latter metric because it allows us to quantify the reduction of OCE workload both at the level of every individual bug and as an aggregate. The former metric, i.e. the amount of time saved, can only provide us an aggregate across all alerts. Moreover, unless we shadow OCEs over an extended period of time, it is difficult to accurately quantify the time saved.

Given a bug, an OCE will investigate an average of $c/2$ commits to localize it, where c is the number of commits in the buggy build. If the OCE uses Orca, they need investigate at most r commits, where r is the rank of the correct commit in the results that the UI shows. If Orca does not find the correct commit, then apart from the 10 commits that Orca shows, the OCE needs to investigate an additional $(c-10)/2$ commits in expectation.

Figure 5 shows the number of commits investigated with and without Orca, for all 48 bugs that we evaluated, as a cumulative distribution function. Over all 48 bugs, without Orca, the OCE investigates a median of 22.75 commits, and an average of 28.9 commits. With Orca, she investigates a median of just 3.5 commits and an average of only 9.4 commits. Therefore, using Orca causes a $6.5\times$ reduction in median OCE workload and a $3\times$ (67%) reduction in the OCE's average workload. For the 37 bugs Orca localizes, this reduction factor in the average is much higher, i.e. $9.7\times$. For the 4 bugs that were caught only because of the build provenance graph, the OCE had to investigate an average of 59.4 commits without Orca, and only 1.25 commits with it. This is a $47.5\times$ improvement. These numbers point out the benefits that the build provenance graph gives us, and the benefits overall of using Orca for commit-level bug localization.

6.3 Performance

Finally, we evaluate the performance of Orca. We run Orca on a 32 core, 2GHz Intel Xeon E7-4820 CPU with 64 GB memory. We ran all 48 queries in sequence to obtain these results. First, we evaluated the effect of the Redis cache on Orca's average query response time. Using

Redis, the average response time reduced from 12.4 seconds to 5.97 seconds, a gain of 51.8%. Next, we varied the degree of parallelism in Orca from 32 to 128 using the C# MAXDOP parameter [37] and noticed a significant effect on average query response time. With parallelism set to 32, the average response time is 30.94 seconds, whereas with parallelism of 128, it is 5.97 seconds. Our evaluation shows that there is a significant potential for parallelizing Orca further, thereby catering to many more queries and providing lower response times. Therefore we can effectively scale Orca out to more services within our enterprise without loss in performance.

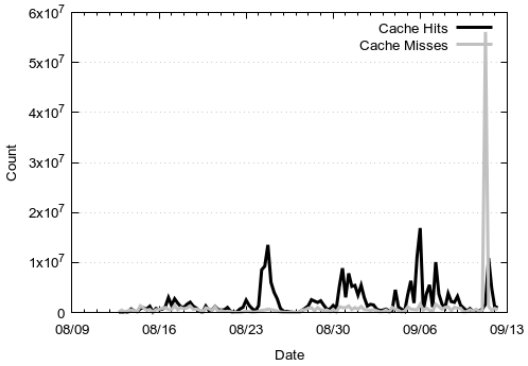

Figure 6: The efficacy of using Redis in Orca's deployment.

Figure 6 shows the effect of using Redis with Orca's deployment over two months, starting from August 2018. On average, the number of hits is 2.2 times the number of misses. This shows that users of the Orca API make queries that have locality and therefore benefit greatly from the use of the Redis cache. This is to be expected as successive queries to Orca will be highly likely for the same builds and therefore will access similar difference sets.

7 Discussion

In this section, we first discuss the generalizability of Orca to other services. We then discuss the limitations, and how we intend to address these in the future.

7.1 Generalizing Orca

Fundamentally, in a CI/CD pipeline, to save time and resouce-usage, services combine various commits before they build, test and deploy. Performing these procedures after every commit would be prohibitively expensive. Since such an aggregation of commits is absolutely necessary, so is the need for a tool that localizes bugs at the commit-level, such as Orca.

Though we describe Orca in the context of a large service and post-deployment bugs, we believe the techniques we have used also apply generically to many Continuous Integration/Continuous Deployment (CI/CD) pipeline. This is based on our experience with multiple services that Orca is operational on within our organization. Orca needs expressive symptoms as input. Modern-day services use rich monitoring systems enabled by infrastructure such as Nagios [33] and AlertSite [2] which can provide probe-level symptoms of problems.

7.2 Future Work

Two types of bugs that Orca currently cannot localize are performance and configuration issues. Addressing these bugs, therefore, are immediate next-steps.

To deal with performance-related bugs, we plan to incorporate better anomaly detection algorithms, and correlate anomalies with code-changes that could potentially cause them. We believe that the principal idea of Orca, i.e. keyword match, can also be applied to bugs that arise from faulty configuration settings. Therefore, we plan to include configuration-based difference sets into our search engine.

8 Conclusion

In this paper, we described Orca and the differential bug localization algorithm. Orca uses differential code analysis and the build provenance graph to find buggy commits in large-scale services. Orca is deployed with a large email and collaboration platform. We have shown that Orca finds the correct buggy commits in about 77% of bugs that we studied, and causes a $3\times$ reduction in the work done by the OCE. We have also shown that Orca is efficient, accurate and easy to deploy.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPlan Notices*, volume 25, pages 246–256. ACM, 1990.

[2] AlertSite - Application Performance Monitoring Tools. `https://smartbear.com/product/alertsite/overview/`. Accessed: 2018-09-25.

[3] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala. Debugadvisor: A recommender system for debugging. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software*

Engineering, ESEC/FSE '09, pages 373–382, New York, NY, USA, 2009. ACM.

[4] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. T. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of FSE*, 2011.

[6] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM systems journal*, 22(3):229–245, 1983.

[7] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 2–9. ACM, 2002.

[8] J. L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[9] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA, 2009. IEEE Computer Society.

[10] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 217–231, Berkeley, CA, USA, 2014. USENIX Association.

[11] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[12] H. Cleve and A. Zeller. Locating causes of program failures. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 342–351. IEEE, 2005.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[14] O. De Moor, M. Verbaere, and E. Hajiyev. Keynote address:. ql for source code analysis. In *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pages 3–16. IEEE, 2007.

[15] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*, pages 313–324, Västeras, Sweden, 2014.

[16] FastTree (Gradient Boosted Trees). `https://docs.microsoft.com/en-us/machine-learning-server/r-reference/microsoftml/rxfasttrees`. Accessed: 2018-08-23.

[17] Git - Version Control System. `https://git-scm.com/`. Accessed: 2018-09-25.

[18] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[19] B. Godlin and O. Strichman. Regression verification. In *Proceedings of the 46th Annual Design Automation Conference*, pages 466–471. ACM, 2009.

[20] A. Hindle, D. M. German, and R. Holt. What do large commits tell us? a taxonomical study of large commits. In *Proceedings of MSR*, 2008.

[21] I. Infonetics. The cost of server, application, and network downtime: Annual north american enterprise survey and calculator. *Computing*, 2016.

[22] M. Isard. Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, Apr. 2007.

[23] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 345–355. ACM, 2013.

[24] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: Opportunities, applications, and challenges. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 201–204, New York, NY, USA, 2010. ACM.

[25] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports

(n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481. IEEE, 2015.

[26] B. Liblit, A. Aiken, M. Naik, and A. X. Zheng. Scalable statistical bug isolation. In *In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 15–26. ACM Press, 2005.

[27] H. P. Luhn. Key word-in-context index for technical literature (kwic index). In *Proceedings of the 136th Meeting of the American Chemical Society, Division of Chemical Literature*, 1959.

[28] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[29] K. L. McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[30] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *ICSE SEIP Track*, 2017.

[31] ML.NET Machine Learning Framework. `https://www.microsoft.com/net/learn/apps/machine-learning-and-ai/ml-dotnet`. Accessed: 2018-08-23.

[32] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of ESEM*, 2007.

[33] Nagios - The Industry Standard In IT Infrastructure Monitoring. `https://nagios.org`. Accessed: 2018-09-19.

[34] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237. ACM, 2008.

[35] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52. ACM, 2011.

[36] M. Rath, D. Lo, and P. Mder. Replication data for: Analyzing requirements and traceability information to improve bug localization, 2018.

[37] M. E. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. Microsoft Press, 6th edition, 2012.

[38] S. Wang and D. Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.

[39] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[40] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 117–132, New York, NY, USA, 2009. ACM.

[41] J.-M. Yang, R. Cai, F. Jing, S. Wang, L. Zhang, and W.-Y. Ma. Search-based query suggestion. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, pages 1439–1440, New York, NY, USA, 2008. ACM.

[42] K. C. Youm, J. Ahn, J. Kim, and E. Lee. Bug localization based on code change histories and bug reports. In *Software Engineering Conference (APSEC), 2015 Asia-Pacific*, pages 190–197. IEEE, 2015.

[43] M. Yu, A. G. Greenberg, D. A. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *In Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[44] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 293–306, Berkeley, CA, USA, 2012. USENIX Association.

[45] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[46] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson. Deepview: Virtual disk

failure diagnosis and pattern detection for azure. In *In Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[47] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 565–581, New York, NY, USA, 2017. ACM.

[48] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. Lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 629–644, Berkeley, CA, USA, 2014. USENIX Association.

[49] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, pages 14–24. IEEE Press, 2012.