MIT Libraries | DSpace@MIT

MIT Open Access Articles

Interactive Production Performance Feedback in the IDE

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Cito, Jurgen, Leitner, Philipp, Rinard, Martin C and Gall, Harald C. 2019. "Interactive Production Performance Feedback in the IDE." Proceedings - International Conference on Software Engineering, 2019-May.

As Published: http://dx.doi.org/10.1109/ICSE.2019.00102

Publisher: IEEE

Persistent URL: https://hdl.handle.net/1721.1/137037

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

Interactive Production Performance Feedback in the IDE

Jürgen Cito
MIT
Cambridge, MA, USA
jcito@mit.edu

Philipp Leitner
Chalmers | University of Gothenburg
Gothenburg, Sweden
philipp.leitner@chalmers.se

Martin Rinard
MIT
Cambridge, MA, USA
rinard@mit.edu

Harald C. Gall
University of Zurich
Zurich, Switzerland
gall@ifi.uzh.ch

***Abstract*—Because of differences between development and production environments, many software performance problems are detected only after software enters production. We present *PerformanceHat*, a new system that uses profiling information from production executions to develop a global performance model suitable for integration into interactive development environments. *PerformanceHat*'s ability to incrementally update this global model as the software is changed in the development environment enables it to deliver near real-time predictions of performance consequences reflecting the impact on the production environment. We build *PerformanceHat* as an Eclipse plugin and evaluate it in a controlled experiment with 20 professional software developers implementing several software maintenance tasks using our approach and a representative baseline (Kibana). Our results indicate that developers using *PerformanceHat* were significantly faster in (1) detecting the performance problem, and (2) finding the root-cause of the problem. These results provide encouraging evidence that our approach helps developers detect, prevent, and debug production performance problems during development before the problem manifests in production.**

***Index Terms*—software performance engineering, IDE, user study**

I. INTRODUCTION

Because of differences between development and production execution environments, many software performance problems are detected only after software enters production [15]. To track down and correct such performance problems, developers are currently faced with the task of inspecting monitoring data such as logs, traces, and metrics and must often run additional performance tests to fully localize the root cause of the problem. Problems associated with this situation include the deployment of software with performance problems, the time required to identify and correct performance problems, and the need for developers to function effectively in two very different environments, specifically the software code view environment during development and the monitoring-based environment during deployment. Previous studies have shown that switching between separate views makes it difficult for developers to maintain a clear image of the overall context of runtime behavior [19], [25]. This separation is particularly problematic when developers are immersed in the context of a particular task [9], [27].

We present a new tool, *PerformanceHat*, that integrates production monitoring information directly into the source code view. Instead of requiring developers to manually analyze monitoring data and perform additional test runs to obtain relevant information about performance problems, *PerformanceHat* works with a performance model derived from monitoring data collected during production runs. As the developer modifies the code, *PerformanceHat* incrementally updates the model to provide the developer with performance feedback in near real time. Because the model is derived from production monitoring data, the performance feedback accurately reflects the performance consequences of specific developer changes in the production environment, not in the development environment. Benefits of this system include the early detection (and correction) of performance problems in development before the software enters production and a tight integration of performance feedback into the development process. We evaluate *PerformanceHat* in the Eclipse IDE with Java. In a controlled experiment with 20 professional software developers, the data show that developers using *PerformanceHat* find the root cause of production performance problems significantly faster than a control group using standard techniques. At the same time, developers using *PerformanceHat* when working on non-performance relevant tasks experienced no drop in productivity compared with the control group.

II. BACKGROUND & RELATED WORK

We introduce some background on profiling and monitoring, discuss related work around software performance prediction, and source code view augmentation.

Profiling vs Monitoring. Profiling is a form of dynamic program analysis that measures performance aspects of a running program. Software performance is highly dependent on the execution environment and workload of a system. Therefore, information provided by profilers executed locally is often not enough to identify performance issues in production environments. A performance monitoring tool continuously monitors components of deployed software systems. It collects several performance metrics (such as response time or CPU utilization) from the monitored application and usually displays them in form of time series graphs in dashboards. A recent study has shown that monitoring tools exhibit enough information to be used to identify performance regressions [1]. Our approach, *PerformanceHat*, utilizes information collected in state-of-the-art monitoring tools to build a performance model that is integrated in the developer workflow in the IDE.

Impact Analysis & Performance Prediction. Change impact analysis supports the comprehension, evaluation, and implementation of changes in software [21], [6]. Most of the work

that is related to change impact analysis and performance operates on an architectural level [16], [5] and is not supposed to be "triggered" during software development. Recent work by Luo et al.[24] uses a genetic algorithm to investigate a large input search space that might lead to performance regressions. Our approach for impact analysis is applied live, i.e. during software development, and leverages an initially build performance model from monitoring data to incrementally reflect software changes to provide early feedback to software developers.

Augmenting Source Code Views. Several papers have proposed augmenting source code views to support program analysis efforts. Hoffswell et al. [17] introduce different kinds of in-situ visualizations related to runtime information in the source code to improve program understanding. Lieber et al. [23] augment JavaScript code in browser debugging tools with runtime information to support reasoning on asynchronous function interaction. Beck et al. augment method definitions in the IntelliJ IDE with in-situ visualizations obtained by sampling stack traces [4].

Our approach goes beyond augmenting local runtime information (e.g., generated by tests) to deal with distinct scalability challenges that we describe in our approach in Section IV-C. Further, we go beyond visualizing runtime information in-situ and provide early feedback on source code changes by leveraging incremental performance analysis.

III. Scope

Our research targets systems with particular properties:

- ***Online Services***: We target online services that are delivered as a service (SaaS applications), typically deployed on cloud infrastructure, and accessed by customers as web applications over the Internet. Specifically, we do not consider embedded systems, "big data" applications, scientific computing applications, or desktop applications. While an approach similar to ours could also be used for a subset of those, the concrete modeling and performance-related challenges would change, taking them out of scope for the present study.
- ***Software Maintenance***: We assume that the application is already deployed to production and used by real customers. However, given the ongoing industrial push to lean development methodologies and continuous deployment [30], many SaaS applications are "in maintenance mode" for the vast majority of their lifetime.
- ***System-Level Performance***: *PerformanceHat* supports performance engineering on a systems level rather than improving, e.g., the algorithmic complexity of a component. Hence, a focus is put on component-to-component interactions (e.g., remote service invocations, database queries), as these tend to be important factors contributing to performance regressions on a system level, while also being hard to test without knowing production conditions.
- ***Production Observable Units***: Our approach is limited to modeling methods that are actively measurable by existing monitoring tools. Thus, methods that might have suboptimal (theoretical) computational complexity, but do not exhibit any significant overhead that is captured by monitoring will not be modeled.

IV. Interactive Monitoring Feedback in the IDE

Our theory is that software developers are enabled to identify and prevent performance issues faster when source code is augmented with monitoring data and developers receive immediate (i.e., near real time) feedback on code changes. We state the goals for our approach, *PerformanceHat*, and describe the models and techniques that allow us to achieve them. To guide the design of our approach, we formulate two goals based on our theory:

- ***Operational Awareness:*** We want to provide *operational awareness* to developers by tightly integrating monitoring data into the development workflow. By that, developers should become more aware of the operational footprint of their source code.
- ***Contextualization:*** At the same time, we aspire to provide *contextualization* of these operational aspects. Monitoring data should be available in the context of current development task to minimize context switches to other external monitoring tools.

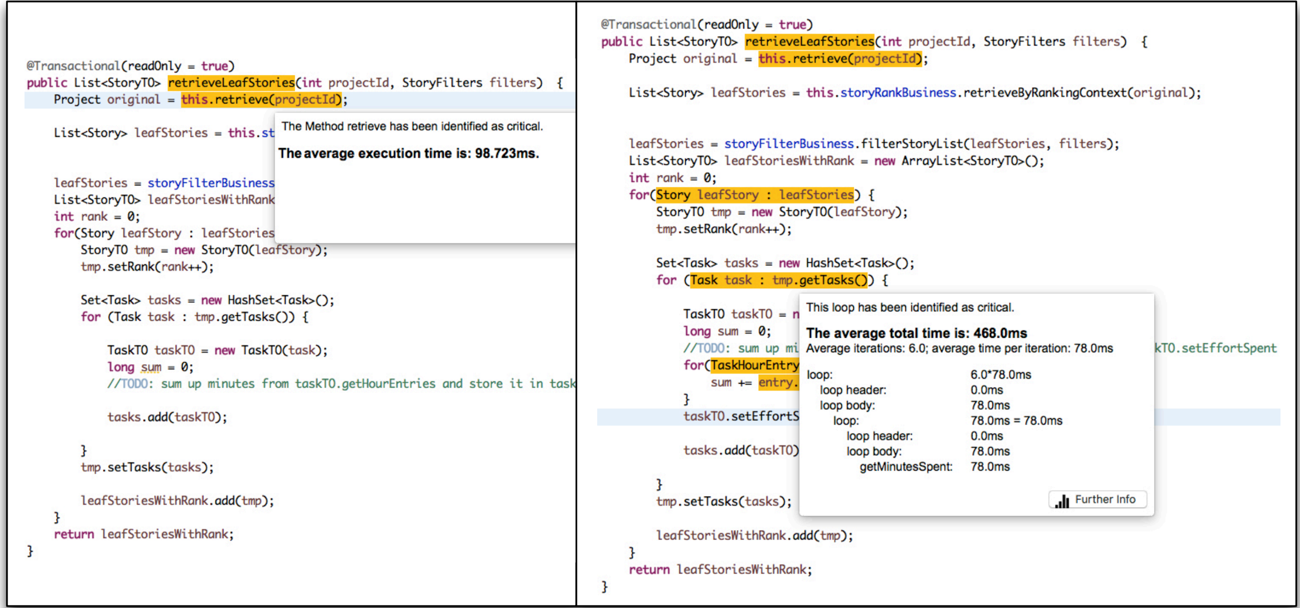
Based on these principles, we implement *PerformanceHat*, as part of the Eclipse IDE with Java (Figure 1). Whenever a class is loaded we construct a performance model and display it as in-situ annotations (yellow highlighting) in the source code. When hovering over these annotations, a box appears that provides performance information from the model retrieved from production monitoring. Performance analysis is built into the incremental build process, i.e., we incrementally update the performance model every time a developers saves new changes (by triggering the incremental build of Eclipse). Through this process we provide interactive performance analysis updates, so that developers retrieve immediate feedback of the impact of their changes.

We now discuss more formally how we achieve our goals by deriving an In-IDE Performance model.

A. In-IDE Performance Model

We construct our model by establishing a relation from programs to datasets retrieved from monitoring tools. We then describe how we can incrementally update our performance model by reflecting and propagating changes in the model.

We consider programs p as syntactically valid programs of a language $\mathbb{P}$. A program $p \in \mathbb{P}$ consists of a set of methods, $m \in M(p)$, where every method m is uniquely identifiable through $id(m)$ (e.g., fully qualified method names in Java) organized in classes (or any other unit of organization for methods, e.g., modules). A syntactically valid program, $p \in P$, can be transformed into an Abstract Syntax Tree, a tree representation of the source code of p, denoted by $AST(p)$. We consider a simplified AST model where the smallest entity nodes are method declarations and statements within a method (method invocations, branching statements, and loops). Formally, an AST is a tuple (A, a_0, ζ), where A



(a) Displaying Operational Footprint

(b) Inferring the Performance of Newly-Written Code

Fig. 1: **(a)** *PerformanceHat* in action displaying execution times in production contextualized on method level. The box is displayed as the developer hovers over the marker on the "this.retrieve" method invocation. **(b)** After introducing a code change, the inference function attempts to predict the newly written code. Further, it is propagated over the blocks of foreach-loops. The box is displayed when hovering over the loop over the `Task` collection, showing the prediction with supporting parameters.

is set of nodes in the AST (*source code artifacts*), $a_0 \in A$ is the root node and $\zeta : A \mapsto A^*$ is a total function that, given a node, maps it to a list of its child nodes. Each node $a_i \in A$ has a unique identifier, $id(a_i)$. For convenience, we also define a function $AST_M(m)$ that returns the AST for a method $m \in M(p)$. Formally, $m_{AST} \subseteq p_{AST}$, where $m_{AST} = AST_M(m), p_{AST} = AST(p)$, $m \in M(p)$ and $id(m) = id(a_0)$ in m_{AST}.

1) Trace Data Model: Our approach relies on execution traces that have been collected at runtime, either through observation, instrumentation, or measurement. While this data could potentially have different types, the focus of this paper is on runtime data that is relevant to software performance (e.g., execution times, workload). Let us consider a dataset $\mathcal{D}$ to be the set of trace data points. The model of a data point $d_i \in \mathcal{D}$ is illustrated in Table I. The illustrative trace shown in the table is an actual trace type used in the evaluation through the monitoring tool Kieker [32]. We model the point in time the data point has been measured or observed and the runtime entity that "produced" the data point (the granularity ranges from methods to API endpoints to operating system processes). We assume every trace has a numerical or categorical value of the observation. Many traces are also associated with some kind of label that is part of the trace meta data (e.g., method name). The context is a set of additional information that signifies, for instance, on which infrastructure the entity was deployed on, or, which log correlation id (in the example it is called "SessionId") was involved [18].

TABLE I: Trace Data Model

Abstract	Description	Illustrative Trace
t	Recorded Time	Logging Timestamp
$\mathcal{E}$	Observed Entity	Java method
$\mathcal{T}$	Trace Type	Execution Time (ET)
$\mathcal{V}$	Primitive Value	Measured time (e.g. 250ms)
L	Label	Method Name (e.g., us.ibm.Map.put)
$\mathcal{C}$	Context	{Stack Size, SessionId, Host,...}

2) Trace Mapping: In an initial step, AST nodes are combined with the dynamic view of runtime traces. This mapping constitutes a relation between nodes in the AST and a set of trace data. On a high level, this process is inspired by previous work in software traceability using formal concept analysis [26], [11], which is a general framework to reason about binary relationships.

A set of traces can be mapped to different AST node types (e.g., method invocations, loop headers) based on different specification criteria in both node and trace. In this particular case of response times, we map data based on the fully-qualified method name in Java, that is both available as part of the AST node and in the trace data. Specifications define declarative queries about program behavior that establish this mapping relationship.

Specification Queries. We model the relation between source code artifacts $a \in A$ in $AST(p)$ to trace data points $d \in \mathcal{D}$ is

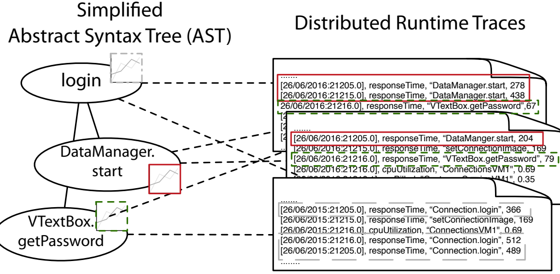


Fig. 2: An illustration of the performance model construction process: Based on a specification function (in this particular case, identity matching), multiple data points from the dataset are mapped to an AST node.

modeled as a mapping $\mathcal{S} : A \mapsto \mathcal{D}^*$. The mapping is directed by a declarative *specification predicate* $SP : A \times \mathcal{D} \mapsto \{true, false\}$. The predicate decides based on an expression on attributes within the source code entity and data point whether there is a match. While the specification can take many different forms depending on the application domain, we illustrate this concept by briefly outlining two typical examples for online services:

- *Entity-Level Specification.* In its simplest form the predicate returns true if information of one trace can be exactly mapped to one source code entity based on an exact attribute matching. Let us again look at our the example of response time traces: $SP_{RT}(a, d) = (id(a) = L(d)) \wedge \tau(d) = ET$. This is a common use case for execution times or method-level counters that can be mapped to method definitions and method invocations. Measurements of multiple threads are attributed to the same method declaration when encountered in the AST.
- *System-Level Specification.* A more complex specification could take the form of mapping memory consumption, which is usually measured at the system or process level, to method invocations through program analysis and statistical modeling. The idea is to sample system-level runtime properties over time and at the same time, through instrumentation, sample how long each method has been executed at recorded points in time. Overlaying both measurements can approximate the impact of certain method invocations to the system level property (e.g., memory consumption). This approach has been already shown to work well when mapping system-level energy consumption on source line level [22].

B. Incremental Performance Analysis

During software maintenance, software developers change existing code to perform change requests. One of our goals is to provide early feedback regarding software performance so that software developers can make data-driven decisions about their changes and prevent performance problems from being deployed to production. To achieve this, we design an incremental analysis for software performance when adding method invocations and loops into existing methods. We focus on these particular changes because existing work by Sandoval et al. [29] has shown that they have the most significant effect on software performance.

Changes to source code are reflected as additions or deletions in the AST. Existing work supports formal reasoning of these changes through tree differencing [13], [14]. While this technique would also be a viable option for our approach to trigger an update in our performance model, we apply a slightly different procedure that enables faster analysis. Since, in addition to static source code, we also have access to a dataset $\mathcal{D}$ and a specification function $\mathcal{S}$, we can distinguish between AST nodes that have data attached to them and new nodes without information. Algorithm 1 presents an overview on a holistic approach that combines (1) constructing a performance model for new classes through mapping, (2) inference of changes, and (3) intra-procedural propagation within the AST of a particular method $m \in M(p)$ in linear time with respect to the number of AST nodes in m:

- We iterate through every node in the method AST in BFS order and attempt to associate data from the trace data set $\mathcal{D}$ to the node through the specification query $\mathcal{S}$.
- If the node cannot be associated with existing data, we assume it to be newly added code and add it to a stack (*toInferNodes*).
- Nodes in the stack without data are iterated and attempted to be inferred by a given inference function Γ. Because we pushed nodes from the stack from the previous BFS iteration "outside in", we now infer nodes "inside out".
- Every newly inferred method is added to a *context* set that is passed to the inference model and can be used for nodes that are higher in the hierarchy. *Example:* Let us assume, we have a new method invocation within a for-loop. The new method invocation is inferred before we reach the loop node, thus, we add its information to the context. As we reach the loop node, the loop inference model can use this newly inferred information in the context to adjust its prediction.
- "Passing up" the context is how we achieve propagation.

Incremental Update through Partial Inference: We integrate our analysis into the build process and, thus, into the development workflow. Hence, a sense of immediacy is required. Existing performance prediction methods range from analytical models [3] to very long running simulation models [5]. To obtain a result in an appropriate time frame, our approach requires an analytical performance inference model. To illustrate how a possible inference model can look like, we combine an analytical model with a learning model inspired by Didona et al. [10]. However, our work is general in the sense that a different analytical model for performance inference could be integrated as well.

We briefly illustrate two models the we implemented in our tool, *PerformanceHat*. Formally, an inference model is a function $\Gamma : A \times A^* \mapsto \mathcal{D}^*$ where $A \in AST(p)$. This function attempts to infer new information based on existing, matched data (i.e., its inference context). Different source code artifact types require different inference models. We

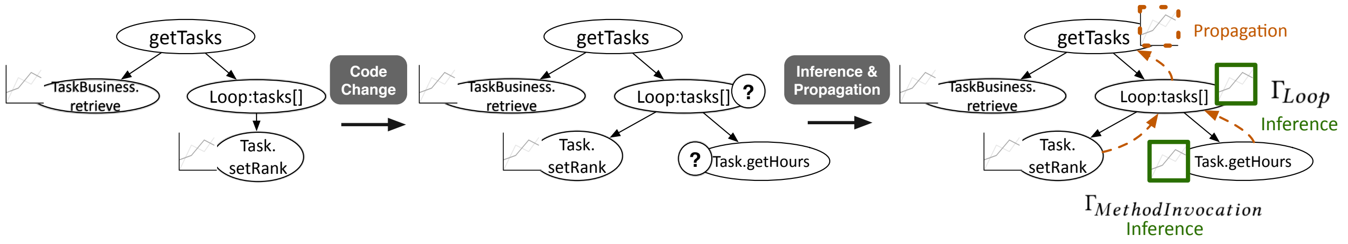


Fig. 3: Sequence of incremental performance analysis and propagation: Code is changed and nodes are inferred from the bottom up and propagated up to the method declaration.

Algorithm 1: Matching, Inferring, and Propagating Runtime Information to AST Nodes in Method m

Data: A method $m \in M(p)$, a dataset $\mathcal{D}$, a specification function $\mathcal{S}$, an inference function Γ

Result: All relevant AST nodes in m annotated with data in $\mathcal{D}$ or with a prediction inferred through Γ

toInferNodes $\leftarrow \emptyset$;

`// Iterator goes through method AST through BFS (i.e. outside-in)`

for *node in* $AST_M(m)$ **do**

  $node.data \leftarrow visit(node, \mathcal{S})$; `// Trigger node-type dependent visitor to match` $\mathcal{D}$ `to node based on` $\mathcal{S}$

  if *not node.data* **then**

    toInferNodes $\leftarrow$ toInferNodes $\cup \{node\}$; `// Adding nodes unknown to` $\mathcal{D}$ `in this context to be inferred`

  end

end

context $\leftarrow \emptyset$;

while *not empty(toInferNodes)* **do**

  currentNode $\leftarrow$ toInferNodes.pop() ; `// Infer information about nodes from the inside out`

  currentNode.data = $\Gamma(currentNode, context)$;

  context $\leftarrow$ context $\cup \{currentNode\}$; `// Newly inferred data is propagated through passing context up`

end

consider addition of method invocations and loops:

Method Invocation Inference $\Gamma_{MethodInvocation}$: The most atomic change we consider for our analysis is adding a new method invocation. To infer new information about the response time of this method invocation in the context of the parent method, we require information about it from an already existing context (i.e., the method being invoked in a different parent method). Further, we want to adjust the response time based on the different workload parameters of the parent method. Thus, we learn a model $M_{WL} : M(p) \times M(p) \mapsto \mathcal{D}^*$ (any viable regression model) that represents how response times of invocations are affected by different workloads WL. From this learned model, we can infer new method invocations as $\Gamma_{MethodInvocation}(m) = M_{WL}(parent(m), m)$, where *parent:* $M(p) \mapsto M(p)$, is a function that returns the parent method of an invocation.

Loop Inference Γ_{Loop}: When adding new loops entirely or adding new method invocations within the body of a loop, we consider a simple, non-parametric regression model (i.e., an additive model) to infer the average execution time of the loop. Let $l \in AST_M(m)$ be a loop node in method $m \in M(p)$. We build an additive model over the mapped or inferred execution times of all statements (method invocations or other blocks) in the loop body of l multiplied by the average number of iterations $\theta_{size}(l)$. More formally, $t(l) = \sum_{n \in \zeta(l)} s_n(n.data)$ is a model of the execution time of the loop body, where the functions s_n are unknown smoothing factors, that we can fit from existing data in $\mathcal{D}$. Thus, $\Gamma_{Loop}(l) = \theta_{size}(l) \times t(l)$. In case of a foreach loop over a collection, the number of iterations θ_{size} can either be retrieved from instrumenting the collections in the production environment or by allowing the software developer to provide an estimate for this parameter.

Figure 3 illustrates incremental analysis with an example:

- A developer changes the code and adds a method invocation (`Task.getHours`) within a loop (over collection of type `Tasks`).
- The nodes of the new method invocation and its surrounding loop do not have any information attached to them.
- First, the newly introduced method is inferred through $\Gamma_{MethodInvocation}$ and attached to the node.
- The new information is propagated and used in Γ_{Loop} to approximate the new loop execution time.
- All new information is then propagated to all nodes up until the method declaration.

Resulting Performance Model: After describing all steps of construction, we formally summarize our performance model as the tuple $\langle A, \mathcal{D}, \mathcal{S}, \Gamma \rangle$, where

- A is a simplified AST (mostly on method level)
- $\mathcal{D}$ is a trace dataset retrieved from a monitoring tool
- $\mathcal{S}$ is a specification query that establishes a relation from AST nodes to traces in the dataset
- Γ is an inference function that employs lightweight prediction models to update information on AST nodes

We now describe the implementation of *PerformanceHat* and discuss scalability concerns.

C. Scalability Design and Implementation

Given that our performance analysis is integrated into the development workflow, it is important that it does not introduce significant delays and interrupt the workflow. While we

presented how to overcome conceptual impediments to enable immediate analysis earlier (i.e., incremental inference and propagation), we now want to discuss design considerations that reflect on architectural scalability of our approach.

We implemented *PerformanceHat* as a combination of an Eclipse plugin for Java and further components that deal with collecting and aggregating runtime performance information. When a program is loaded in the IDE, we construct our initial performance model. *PerformanceHat* then hooks our incremental performance analysis into Eclipses incremental builder. This means, every time a file is saved, we start the analysis for that particular file.

In initial versions of *PerformanceHat*, every build process triggered fetching new data from a remote instance (e.g., the performance monitoring server), introducing network latency as a bottleneck. An iterative process to improve scalability resulted in the the high-level architecture depicted in Figure 4. It bundles application logic for performance model construction (i.e., specification and inference) as an extension to the IDE, and retrieves data from a local component (*local feedback handler*) that synchronizes with a remote component (*deployed feedback handler*, e.g., a monitoring tool server) located closer to the running application in production.

We provide a brief overview on the efforts that were required to enable scalability to construct and incrementally update a performance model in the IDE:

- *Local Feedback Handler*: Feedback handler is implemented as a Java application, exposing a REST API, with a document data store (MongoDB). The deployed feedback handler is installed on the remote infrastructure, close to the deployed system and has an interface to receive or pull runtime information from monitoring systems (after transforming it into a local model that is subsequently understood by the extension in the IDE). The local feedback handler runs as a separate process on the software developer's local workstation. The reason for this split is that constructing the performance model and performing incremental analysis in the IDE requires fast access to monitoring data. The local feedback handler deals with periodically fetching data from potentially remote systems over an HTTP interface.
- *Local IDE Cache*: Additionally, we reduce the feedback loading latency by introducing a local IDE cache, so that each entry has to only be retrieved once per session. We used an LRU cache with a size of maximal 10000 entries and a timeout of 10 minutes (after 10 minutes the cache entry is discarded, however both these entries are configurable). This reduced build time significantly as discussed in Section VI.
- *Bulk Fetching*: A significant improvement also occurred when, for an empty cache, we registered all nodes that required information from the feedback handler and then loaded all information in bulk.

The prototype implementation, including documentation, is available as an open source project on GitHub [7].

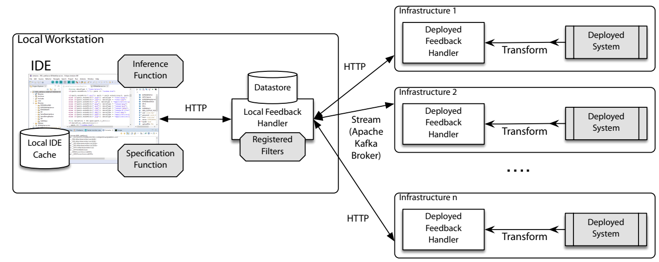


Fig. 4: Resulting scalable architecture for *PerformanceHat*

V. User Study Evaluation

To evaluate whether the proposed approach has a significant impact on decisions made in the course of software maintenance tasks, specifically related to performance, we conduct a controlled experiment with 20 professional software developers as study participants. In the following, we describe the study in detail, outlining our hypotheses, describing programming tasks, measurements and the study setup. We then present the results of the study and discuss threats to validity.

A. Hypotheses

The goal of our study is to empirically evaluate the impact our approach has on software maintenance tasks that would introduce performance issues. To guide our user study, we formulate the following hypotheses based on this claim.

> $H0_1$: Given a maintenance task that would introduce a performance problem, software developers using our approach are *faster in detecting the performance problem*

We are interested in knowing whether the presence of performance data and inference on code level supports software developers in detecting performance issues in code faster.

> $H0_2$: Given a maintenance task that would introduce a performance problem, software engineers using our approach are *faster in finding the root cause of the performance problem*

Additionally, when a high level problem has been detected, we are interested to see whether our approach allows software developers to find the root cause of the issue faster (i.e., does it improve debugging of the performance issue).

> $H0_3$: Given a maintenance task that is *not relevant to performance*, software engineers using our approach are *not slower than the control group in solving the task*

As not all software maintenance tasks potentially introduce a performance problem, we are equally interested whether our approach introduces overhead into the development process and thus increases development time.

B. Study Design Overview

The broad goal of our experiment is to compare the presence of our approach to a representative baseline that illustrates how software developers currently deal with handling performance problems in industry. Investigating performance regressions from production is done through performance monitoring tools. We considered using a conventional, local profiler as a baseline, but discarded that option because we want to investigate scenarios that are caught using production workloads. Thus, local workloads generated by test suites would not surface in the local profiling tool (e.g., VisualVM) and would also not reflect the reality of how these kinds of performance issues are identified [1], [8].

We design our user study as a controlled experiment using a between-subject design, a common approach in empirical software engineering studies [33], [12], [28]. In between-subject design, study participants are randomly assigned in one of two groups: a treatment group and a control group. Both groups have to solve the same programming tasks. The control group uses a common tool to display runtime performance information, Kibana, in combination with Eclipse to solve the tasks. The treatment group uses our approach within Eclipse to solve the tasks. As the study application we make use of Agilefant, a commercial project management tool whose source code is entirely available as open source, because it represents a non-trivial industrial application that exhibits real life performance issues, which have already been discussed in previous work [24].

C. Study Participants

Many empirical software engineering studies rely on students as study participants to evaluate their approaches. For our approach, however, this was not an option as our study requires understanding and experience of runtime performance issues which are usually only encountered when deploying production software. Thus, we recruited 20 study participants from 11 industrial partners through snowball sampling [2]. The study participants have at least 2 years of professional software development experience and who have previously worked with Java. Performance skill level was self-assessed by the participants based on a Likert-scale question on their software performance ability. We equally distributed the participants between control and treatment group based on their experience and performance skill level.

D. Programming Tasks and Rationale

When designing programming tasks for controlled experiments in software engineering research, we are faced with the trade-off of introducing a realistic scenario and minimizing task complexity and duration to properly capture the effects between the groups and avoiding unnecessary variance [33]. With Agilefant as our study application, we aim for a more realistic scenario in an industrial application. We introduce two types of tasks in the controlled experiment. We present the study participants with software maintenance tasks that are relevant to software performance, but also with tasks that do not have an impact on performance. The rationale for mixing the task types has two particular reasons, which are also reflected in our hypotheses. First, we want to understand to what extent the augmentation of source code with performance data in our approach is a distraction that introduces additional cognitive load into tasks not relevant to performance (see $H0_3$). Second, we want to avoid learning effects after initial tasks in study participants (i.e., them knowing that looking at performance data is usually a way to solve the task). We now give a brief description of the tasks and types used in the study. We briefly describe the tasks (T1 to T4) in the text below. A more detailed description of the tasks with corresponding source code can be found in our online appendix[1].

Performance Relevant Tasks (T2 and T4): Work by [24] discovered code changes in our case study application that lead to performance problems. We extracted two relevant change tasks from these changes for T2 and T4. In T2, the study participants retrieve a collection from a method within a field object to extract object ids and add them to an existing set in a loop. However, this method is quite complex and introduces a performance problem. The participants need to investigate the issue over multiple class files and methods and reason over performance data to find the root cause of the performance problem.

T4 requires the study participants to iterate over an existing collection to retrieve a value and compute a summary statistic, that should then be attached to the parent object. The method to retrieve the lower-level value is lazily loaded and thus slower than maybe expected. Additionally, the new code is located within two nested for-loops. Participants need to retrieve performance information on all newly introduced statements and then reason about the propagation up to the method definition.

Non-Performance Tasks (T1 and T3): We designed the regular tasks (non-performance tasks) to be non-trivial, i.e., that a performance problem might hide in the added statements. For T1, study participants need to set a particular object state based on a value retrieved from a method call on an object passed as a parameter (in which the underlying computation is unknown). For T3, the study participants need to iterate over a collection and compute the sum of a value that needs to be attached to a transfer object (similar to T4).

E. Measurements

In the experiment, we performed a number of different measurements, depending on the task type. For every task, we measure a total time required to solve the task. For *performance relevant tasks*, we distinguish two more measurements:

Total Time (T): We measure the time it takes our study participant to solve the task. Beware that we only start measuring when the participants signaled that they understood the task and navigated to the correct location in the code to conduct the maintenance task. We decided for this protocol to avoid measuring the time it takes for task comprehension and task

[1] http://sealuzh.github.io/PerformanceHat/

navigation (which is not the aim of our research and would introduce unnecessary variance into our experiment).

First Encounter (FE): For tasks involving performance problems, we measure the first encounter of the study participant with the realization that a performance problem has been introduced. This realization can come through inspecting performance data to the newly introduced artifact (either in Kibana by the control group, or in the IDE with our approach in the treatment group) or by attempting to deploy the new code and receiving feedback from performance tests (see Section V-F for details on the setup).

Root-Cause Analysis (RCA = T - FE): Starting from the time of the first encounter of the introduced performance problem (FE), we measure the time until the participant finds the root cause of the performance problem (RCA). We consider the root cause found when the participant can point to the lowest level artifacts (i.e., method invocations) available in the code base that are the cause for the degraded performance effect, and can back their findings with performance data. Performance data can be queried through the provided tools in each group.

F. Study Setup

We conducted the experiments on our own workstation, on-site with each study participant. We now describe the technical environment for the experiments and the protocol.

Environment: The study application was deployed within Docker containers on our own workstation. Performance data was collected through the performance monitoring tool Kieker [32]. For the control group, we also deployed the ELK stack [20], a common setup that collects distributed log files with Logstash, stores them centrally in the database ElasticSearch, to finally display them in the dashboard/visualization tool Kibana. The participants in the control group solely interact with Kibana. Since the standard setup for Kibana only displays the raw form of the collected logs, we provided standard dashboards for the participants that displayed average execution times for each method, which is the same information provided by our approach in the treatment group. For the treatment group, we deploy the *feedback-handler* component that pulls performance data from Kieker directly and converts them into our model with an adapter. The participants were given a standard version of Eclipse Neon in version 4.6.1, which included a text editor and a treeview and no further installed plugins (except, of course, our approach in the treatment group). Participants were able to "hot-deploy" the classes of single tasks separately by executing a console script. When executing the script, performance tests relevant to the task were simulated and the participants were given feedback whether their changes introduced a performance problem and to what extent (response time in seconds).

Protocol: In the first 15 to 20 minutes, the study participants were given an introduction into our study application and its data model, the provided tools, and into the task setting. If needed, the participants were given an introduction to the Eclipse IDE. The control group was given an introduction to Kibana and how it can be used in the experiment setting to potentially solve the programming tasks. The same was done for our approach with the treatment group. Over the course of solving the tasks, participants were encouraged to verbalize their thoughts (i.e., "think-aloud" method). All sessions were recorded for post-analysis with consent of the study participants. Participants were given thorough task descriptions and were encouraged to ask questions to properly understand the questions. When participants signaled that they understood the task and they started programming, we started collecting our measurements. After completing all tasks, we debriefed the participants and asked about their study experience and collected feedback about content and process.

G. Study Results

Table II shows the mean and standard deviation of results for all tasks and measurements, grouped in control and treatment group. We first use the Shapiro-Wilk test to test whether our samples come from a normally-distributed population. We were not able to verify the normality hypothesis for our data. Thus, we perform a Mann Whitney U test, a non-parametric statistical test, to check for significant differences between the population of the two groups and Cliff's delta to estimate the effect size, similar to other comparable works in empirical software engineering [12], [28].

TABLE II: Results (in seconds) over all tasks and measures for treatment and control presented as "Mean ($\pm$ Standard Deviation)", together with the p-value resulting from Mann Whitney U statistical tests. FE and RCA are only given for the performance relevant tasks T2 and T4. P-values < 0.05 are marked with.

	Treatment	**Control**	**Mann Whitney U**
T1 (Total)	231.8 ($\pm$ 71.61)	232 ($\pm$ 84.1)	0.7614
T2 (Total)	267 ($\pm$ 65.35)	464 ($\pm$ 76.61)	*0.0001
T2 (FE)	153.3 ($\pm$ 49.76)	222.7 ($\pm$ 46.69)	*0.0125
T2 (RCA)	113.7 ($\pm$ 40.72)	241.3 ($\pm$ 66.05)	*0.0001
T3 (Total)	239.8 ($\pm$ 57.22)	211.2 ($\pm$ 68.63)	0.1853
T4 (Total)	212.4 ($\pm$ 43.33)	288 ($\pm$ 69.88)	*0.0125
T4 (FE)	134.6 ($\pm$ 37.9)	161.8 ($\pm$ 56.63)	0.3843
T4 (RCA)	77.8 ($\pm$ 26.23)	126.2 ($\pm$ 36.13)	*0.0089

The descriptive statistics suggest that the treatment group requires less time to complete each performance relevant task (*Total* measurements, Task 2 and 4). To ease the reading of the empirical measurements, Figure 5 presents the experiment results in form of box plots comparing the time required by control and treatment group over all task and measures side-by-side. In the following, we investigate the results of the experiment with respect to our formulated hypotheses. To avoid losing perspective in further aggregation, we analyze the tasks relating to our hypotheses separately.

Timing in Performance Relevant Tasks ($H0_1$ and $H0_2$): Looking at performance relevant tasks (T2 and T4), both in Table II and Figure 5, the treatment group performs better (in absolute terms) for all measurements. For both total times, the difference is significant (p-value < 0.05, Effect Sizes/Cliff's delta: 0.92 and 0.67). We now go into more detail between both measures for performance relevant tasks:

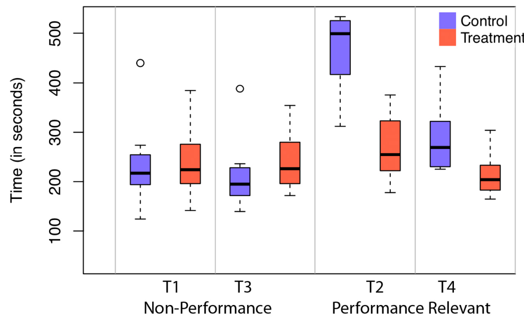


Fig. 5: Total times spent on individual tasks for non-performance tasks (T1 and T3) and performance relevant tasks (T2 and T4) in Control (Kibana) and Treatment (*Performance-Hat*) group.

Detecting Performance Problems ($H0_1$): For the FE (First Encounter) measures (see Figure 6), we see a significant difference in T2/FE (Effect Size/Cliff's delta: 0.92). In T4/FE, however, the difference is not significant. A possible explanation for this difference lies in the structure of the task T4 (see Section V-D). In T4, the code change occurs already in two nested loops. So, even without direct presence of performance data in the process, a software developer can easily speculate that introducing yet another loop leads to an $\mathcal{O}(n^3)$ time complexity. In T2, however, the introduced performance problem was not as obvious by simply inspecting code without performance data.

Root Cause Analysis ($H0_2$): For the measure RCA (Root Cause Analysis – see Figure 6), both T2/RCA and T4/RCA show significant differences (Effect Sizes/Cliff's delta: 0.68 and 0.92) between treatment and control group. Even in the case of T4, where the first encounter was more easily attainable through code inspection alone, the analysis did require querying performance data to pinpoint the root cause of the performance problem.

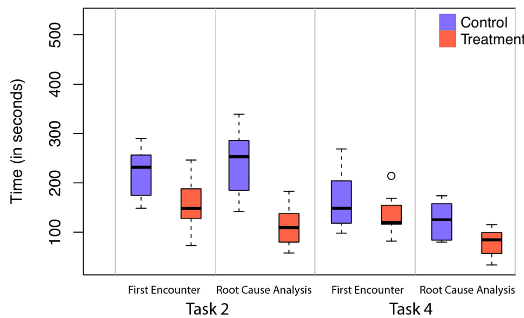


Fig. 6: Times in performance relevant tasks broken down into first encounter of performance problem (FE) and root cause analysis (RCA).

Overhead in Non-Performance Tasks ($H0_3$): For both regular (non-performance) maintenance tasks, T1 and T3, we were not able to reject the null-hypothesis. Beware, that this only means not enough evidence is available to suggest the null-hypothesis is false at the given confidence level. Thus, we have a strong indication that there are no significant differences between the treatment and control group for these tasks. In the context of our study, this is an indication that our approach does not introduce significant cognitive overhead that "distracts" software developers from regular maintenance tasks.

H. Threats to Validity

External Validity refers to threats to the generalizability of the presented results. In this regard, our user study has a threat to external validity with regards to our selection of study participants. Given that participation in such a study is necessarily voluntary, it is possible that our study participants are not representative of the general population of developers of cloud applications. Another issue is our usage of the snowballing sampling technique, which may lead to an overly homogeneous participant population. We have mitigated this threat by carefully supervising our participant demography, and ensuring that, e.g., participants are not from the same company or close circle of collaborators.

Internal Validity describes the extent to which conclusions are justified by the data. We are aware of two major threats to the internal validity of the user study. Firstly, given the complex domain of our approach, some aspects of our study setup were necessarily artificial (i.e., participants did not commit to a real production environment, monitoring data came from a pre-established feedback dataset). Secondly, given the nature of our study, participants were aware of, or could at least suspect, that the study was related to performance. This may have influenced their behavior to be more careful regarding performance. Consequently, it is possible that the effect of *PerformanceHat* outside of a study setting may be more pronounced.

VI. IDE Overhead Analysis

Our performance analysis in *PerformanceHat* is hooked into the incremental build process of the IDE. This means every time a file is saved, the analysis process is triggered. Anecdotally, none of our study participants remarked upon any visible delays introduced through our analysis. However, to gain a more formal picture of its overhead, we numerically study the build time impact of *PerformanceHat* with two case studies. While we deem a small overhead (i.e., increase in build times in the IDE) unavoidable, we need to study whether the additional effort for constructing the performance model (i.e., loading production data, matching to the AST) and rendering warnings and in-situ visualizations does not unduly slow down the IDE.

a) Experiment Setup: We analyze the build time impact in four different experimental settings, as shown in Table III. We use two different case study applications: (1) Agilefant [31], a commercial project management tool whose source code is entirely available as open source, and (2) an existing research prototype. For both applications, we established a

baseline of production feedback data by generating a representative workload on the deployed code. For both applications, we evaluate full project builds (i.e., a build of all Java files) as well as an incremental build that re-builds only a single file (`fi.hut.soberit.agilefant.model.Story` in the case of Agilefant, and a Web service controller in the case of the research prototype). We find these applications interesting, as they represent two orthogonal but typical use cases: a large, monolithic web application in case of Agilefant, and a small service in a much larger composite system in case of the research prototype. Table III also lists the total number of AST nodes for each setting after applying AST construction as in Section IV. Note that these are the simplified ASTs and not the entire ASTs of the original Java source code file.

TABLE III: Summary of experimental settings.

	Application	Build Type	AST Nodes
1	Agilefant	Full Build	271212
2	Agilefant	Incremental Build (Story)	1750
3	Research Prototype	Full Build	731
4	Research Prototype	Incremental Build (Controller)	158

For each of those settings, we further evaluate four different scenarios: with or without cache as discussed in Section IV-C, and with a cold or warm build environment. We refer to the build environment as “cold” directly after (re-)starting the IDE, and as “warm” after the respective build has run at least once. We have chosen these two parameters as preliminary experimentation has shown that both—the cache and whether the IDE has already executed the same build before—have a significant impact on the total build time as well as the build time impact of *PerformanceHat*.

We executed all experiments on a Lenovo ThinkPad X1 laptop, with an Intel Core i5-6200U CPU using a clock speed of 2.40GHz and 8 GB of RAM. The laptop was running Eclipse Neon in version 4.6.1 as the IDE. Background applications and system services have been disabled to the extent possible for the duration of our measurements. We repeated each individual experiment (i.e., each of the settings in every scenario) 5 times to account for natural variability in build times.

b) Results: Figure 7 depicts the total build overhead in seconds in boxplots introduced by *PerformanceHat* in all settings and for all scenarios. Note that the y-axis scale differs for all four settings. This is to be expected, as incremental builds are substantially faster than full builds. Similarly, builds of the large and monolithic Agilefant application take much longer than builds of the research project.

We observe that the overhead for incremental builds never exceeds 2 seconds even in the worst case (no cache, cold IDE, large code base). Using caching alone, the overhead can be reduced to 1.25 seconds. After the IDE is warmed up (which we expect to be the most common scenario in practical usage), the build overhead of *PerformanceHat* for incremental builds becomes completely negligible. For full builds, the overhead depends on the size of the project. For the research prototype, even in case of a full build, the overhead is around 2 seconds in the worst case. For the monolithic Agilefant application,

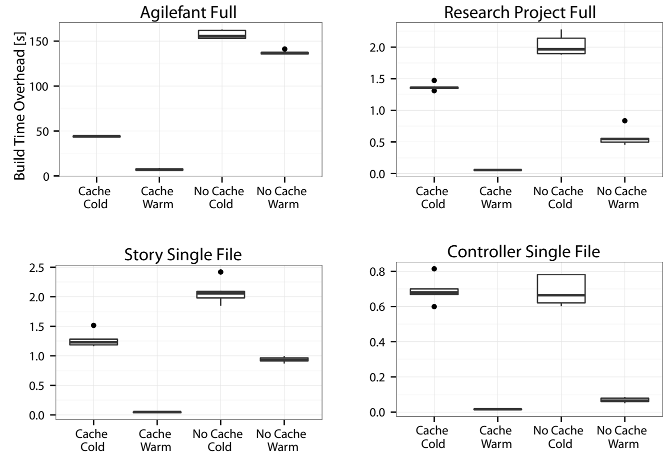


Fig. 7: Total build time overhead of all settings and scenarios in seconds.

PerformanceHat introduces a significant overhead of up to 150 seconds in the worst case for full builds. However, by introducing caching, this overhead can be reduced to less than 50 seconds. And generally, a full build is usually not done more than once per start of the workstation.

We conclude that *PerformanceHat* only introduces a substantial overhead on the total build time in the absolute worst case. This overhead depends on the size of the code base, on the availability of a cache, on whether the IDE is “warm”, and, most importantly, on whether a full project build or an incremental file build is conducted. In the most important use case (incremental builds with cache and warm IDE) the introduced overhead is negligible and not noticeable to developers.

VII. Conclusion

We present a system, *PerformanceHat*, that contextualizes the operational performance footprint of source code in the IDE and raises awareness of the performance impact of changes. Our results from a controlled experiment with 20 practitioners working on different software maintenance tasks indicate that developers using *PerformanceHat* were significantly faster in detecting and finding the root-cause of performance problems. They additionally indicate that when working on non-performance relevant tasks, they did not perform significantly different, illustrating that *PerformanceHat* does not lead to unnecessary distractions.

When designing and researching programming experience, we want to lower cognitive overhead in the software development process. Thus, in the future, we want to introduce “smart thresholds” that learn normal behavior from production and adjust the visibility of performance in code continuously. Further, we want to tackle the potential problem of early optimization by introducing different usage profiles for our approach, that show a different amount and level of detail (e.g., *debugging* vs *design* profile).

REFERENCES

[1] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: an Experience Report. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, pages 1–12, 2016.

[2] R. Atkinson and J. Flint. Accessing Hidden and Hard-to-Reach Populations: Snowball Research Strategies. *Social Research Update*, 33(1):1–4, 2001.

[3] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: a Survey. *IEEE Transactions on Software Engineering (TSE)*, 30(5):295–310, 2004.

[4] F. Beck, O. Moseler, S. Diehl, and G. D. Rey. In Situ Understanding of Performance Bottlenecks Through Visually Augmented Code. In *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC)*, pages 63–72, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[5] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[6] S. Bohner and R. S. Arnold. *Software change impact analysis*. IEEE Computer Society Press, Los Alamitos, Calif, 1996.

[7] J. Cito. Performancehat eclipse plugin. https://github.com/sealuzh/PerformanceHat, 2019.

[8] J. Cito, P. Leitner, T. Fritz, and H. C. Gall. The making of cloud applications: An empirical study on software development for the cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 393–403, New York, NY, USA, 2015. ACM.

[9] M. Csikszentmihalyi. *Flow and the foundations of positive psychology*. Springer.

[10] D. Didona, F. Quaglia, P. Romano, and E. Torre. Enhancing performance prediction robustness by combining analytical modeling and machine learning. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 145–156. ACM, 2015.

[11] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on software engineering*, 29(3):210–224, 2003.

[12] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik. How do api documentation and static typing affect api usability? In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 632–642. ACM, 2014.

[13] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 313–324, 2014.

[14] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering (TSE)*, 33(11), 2007.

[15] J. Hamilton. On designing and deploying internet-scale services. In *Proceedings of the 21st Conference on Large Installation System Administration Conference*, LISA'07, pages 18:1–18:12, Berkeley, CA, USA, 2007. USENIX Association.

[16] C. Heger and R. Heinrich. Deriving work plans for solving performance and scalability problems. In *Computer Performance Engineering*, pages 104–118. Springer, 2014.

[17] J. Hoffswell, A. Satyanarayan, and J. Heer. Augmenting code with in situ visualizations to aid program understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 532. ACM, 2018.

[18] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou. Understanding customer problem troubleshooting from storage system logs. In *Proccedings of the 7th Conference on File and Storage Technologies (FAST)*, pages 43–56, Berkeley, CA, USA, 2009. USENIX Association.

[19] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12):971–987, 2006.

[20] A. Lahmadi and F. Beck. Powering monitoring analytics with elk stack. In *9th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2015)*, 2015.

[21] S. Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 41–50. ACM, 2011.

[22] D. Li, S. Hao, W. G. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–89. ACM, 2013.

[23] T. Lieber, J. R. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2481–2490. ACM, 2014.

[24] Q. Luo, D. Poshyvanyk, and M. Grechanik. Mining performance regression inducing code changes in evolving software. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, pages 25–36, New York, NY, USA, 2016. ACM.

[25] C. Parnin and S. Rugaber. Resumption strategies for interrupted programming tasks. *Software Quality Journal*, 19(1):5–34, 2011.

[26] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 37–48. IEEE, 2007.

[27] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *null*, page 281. IEEE, 2003.

[28] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini. On the positive effect of reactive programming on software comprehension: An empirical study. *IEEE Transactions on Software Engineering (TSE)*, 2017.

[29] J. P. Sandoval Alcocer, A. Bergel, and M. T. Valente. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE)*, pages 37–48. ACM, 2016.

[30] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. Continuous deployment at facebook and oanda. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*, pages 21–30, New York, NY, USA, 2016. ACM.

[31] J. Vähäniitty and K. T. Rautiainen. Towards a conceptual framework and tool support for linking long-term product and business planning with agile software development. In *Proceedings of the 1st international workshop on Software development governance*, pages 25–28. ACM, 2008.

[32] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 247–248, New York, NY, USA, 2012. ACM.

[33] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.