Context-Based Analytics - Establishing Explicit Links between Runtime Traces and Source Code

Jürgen Cito*, Fábio Oliveira†, Philipp Leitner*, Priya Nagpurkar†, Harald C. Gall*
*Department of Informatics, University of Zurich, Switzerland
{cito, leitner, gall}@ifi.uzh.ch
†IBM T.J. Watson Research Center, Yorktown Heights NY, USA
{fabolive, pnagpurkar}@us.ibm.com

***Abstract*—Diagnosing problems in large-scale, distributed applications running in cloud environments requires investigating different sources of information to reason about application state at any given time. Typical sources of information available to developers and operators include log statements and other *runtime information* collected by monitors, such as application and system metrics. Just as importantly, developers rely on information related to changes to the source code and configuration files *(program code)* when troubleshooting. This information is generally scattered, and it is up to the troubleshooter to inspect multiple *implicitly-connected* fragments thereof. Currently, different tools need to be used in conjunction, e.g., log aggregation tools, source-code management tools, and runtime-metric dashboards, each requiring different data sources and workflows. Not surprisingly, diagnosing problems is a difficult proposition. In this paper, we propose Context-Based Analytics, an approach that makes the links between runtime information and program-code fragments *explicit* by constructing a graph based on an application-context model. *Implicit* connections between information fragments are *explicitly* represented as edges in the graph. We designed a framework for expressing application-context models and implemented a prototype. Further, we instantiated our prototype framework with an application-context model for two real cloud applications, one from IBM and another from a major telecommunications provider. We applied context-based analytics to diagnose two issues taken from the issue tracker of the IBM application and found that our approach reduced the effort of diagnosing these issues. In particular, context-based analytics decreased the number of required analysis steps by 48% and the number of needed inspected traces by 40% on average as compared to a standard diagnosis approach.**

Thus

page6

called CloudWave European project

***Keywords*-DevOps; Software Analytics; Runtime Information**

I. Introduction

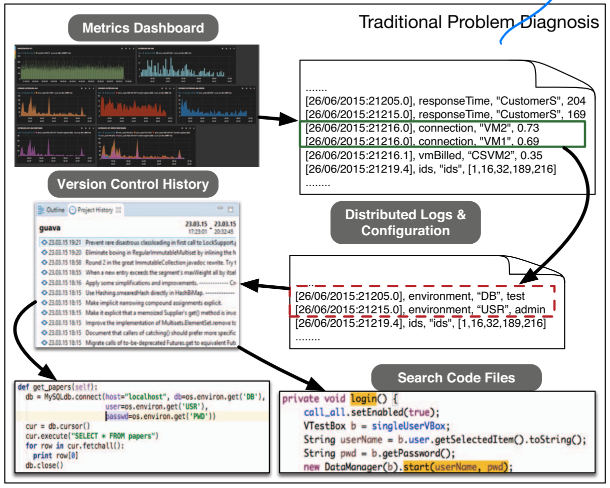
The scalable and ephemeral nature of infrastructure in cloud software development [8] makes it crucial to constantly monitor applications to gain insight into their runtime behavior. Data collection agents send application and infrastructure logs and metrics from cloud guests and hosts to a centralized storage from which all data can be searched and a variety of dashboards are populated.

The ever-increasing need for rapidly delivering code changes to satisfy new requirements and to survive in a highly-competitive, software-driven market has been fueling the adoption of DevOps practices [3] by many companies. By breaking the well-known barrier between development and operations teams, the cultural changes, methodologies, and automation tools brought about by the DevOps phenomenon allow teams to continuously deploy new code to production in cloud environments. It is not uncommon for many companies to deploy new code several times per day [29].

It's common for many companies

However, when a cloud application faces a problem in production, causing a partial or total outage, this fast code-delivery cycle is suddenly halted. Paradoxically, this extreme agility in deploying new code could potentially slow down the continuous delivery cycle, as problems might happen more often, and take longer to resolve, the faster new code is deployed. Hence, it is paramount that developers and operators are empowered to quickly determine the root cause of problems and fix them. Diagnosing a problem invariably always requires analyzing all the information collected from the cloud as well as from tools to manage the application lifecycle into a centralized storage. Typically, the troubleshooter has two choices: look at a variety of dashboards to try to understand the problem, or directly query the data. The vast amount of data collected from all distributed components of a cloud application, spanning several hosts and possibly different data centers, is overwhelming. The *runtime information* gathered includes: log statements; snapshots of cloud guests' state; system metrics, e.g., info CPU utilization and memory consumption; and application metrics, e.g., response time and session length.

Since problems that occur in production often have their root in development-time decisions and bugs [16]—especially if new code is deployed frequently without appropriate test coverage—runtime data needs to be correlated to development-time changes for troubleshooting. However, actually doing so is challenging for developers [8], and involves inspecting multiple fragments of disperse information. An example is depicted in Figure 1a. After becoming Once we notice there is aware of a problem via an alert, a developer investigates and manually correlates logs and metrics, until she finds that a specific change seen in the version control system is the likely culprit. The developer still needs to look at the change and start debugging the code. This traces procedure requires knowledge of tools and processes to obtain information, and perseverance to identify relevant fragments from heterogeneous data sets. Furthermore, when exploring the broader context of a problem, developers build mental models that incorporate source code and other collected information that are related to each other [20]. The mental model consists of different information fragments, derived from the let's take a look at a ~~highlevel~~ brief overview of the proposed context-based analytics


Traditional Problem Diagnosis
Metrics Dashboard
Version Control History
Distributed Logs & Configuration
Search Code Files

```
.......
[26/06/2015:21205.0], responseTime, "CustomerS", 204
[26/06/2015:21215.0], responseTime, "CustomerS", 169
[26/06/2015:21216.0], connection, "VM2", 0.73
[26/06/2015:21216.0], connection, "VM1", 0.69
[26/06/2015:21216.1], vmBilled, "CSVM2", 0.35
[26/06/2015:21219.4], ids, "ids", [1,16,32,189,216]
........
```

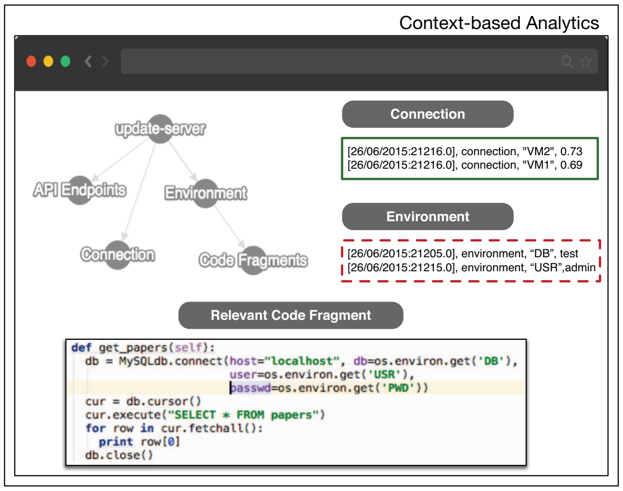
```
guava                          23.03.15 17:23:01 - 23.03.15 20:32:45
23.03.15 19:21  Prevent rare disastrous classloading in first call to LockSupport
23.03.15 19:20  Eliminate boxing in RegularImmutableMultiset by inlining the h
23.03.15 18:58  Round 2 in the great ImmutableCollection javadoc rewrite. Try t
23.03.15 18:55  When a new entry exceeds the segment's maxWeight all by itsel
23.03.15 18:16  Apply some simplifications and improvements. ------------ Cr
23.03.15 18:15  Use Hashing.smearedHash directly in HashBiMap. ------------
23.03.15 18:15  Make implicit narrowing compound assignments explicit.
23.03.15 18:15  Make it explicit that a memoized Supplier's get() method is invo
23.03.15 18:15  Improve the implementation of Multisets.ElementSet.remove tc
23.03.15 18:15  Document that callers of catching() should prefer more specific
23.03.15 18:15  Migrate calls of to-be-deprecated Futures.get to equivalent Fut
```

```
...
[26/06/2015:21205.0], environment, "DB", test
[26/06/2015:21215.0], environment, "USR", admin
[26/06/2015:21219.4], ids, "ids", [1,16,32,189,216]
........
```

```
def get_papers(self):
  db = MySQLdb.connect(host="localhost", db=os.environ.get('DB'),
                       user=os.environ.get('USR'),
                       passwd=os.environ.get('PWD'))
  cur = db.cursor()
  cur.execute("SELECT * FROM papers")
  for row in cur.fetchall():
    print row[0]
  db.close()
```

```
private void login() {
    call_all.setEnabled(true);
    VTestBox b = singleUserVBox;
    String userName = b.user.getSelectedItem().toString();
    String pwd = b.getPassword();
    new DataManager(b).start(userName, pwd);
```

(a) Traditional problem diagnosis requires collecting fragmented information through a multitude of tools and data sources. A hint in one fragment leads to information in another, and so on.



```
[26/06/2015:21216.0], connection, "VM2", 0.73
[26/06/2015:21216.0], connection, "VM1", 0.69
```

```
[26/06/2015:21205.0], environment, "DB", test
[26/06/2015:21215.0], environment, "USR",admin
```

```
def get_papers(self):
  db = MySQLdb.connect(host="localhost", db=os.environ.get('DB'),
                       user=os.environ.get('USR'),
                       passwd=os.environ.get('PWD'))
  cur = db.cursor()
  cur.execute("SELECT * FROM papers")
  for row in cur.fetchall():
    print row[0]
  db.close()
```

(b) Context-based Analytics organizes the plethora of runtime information in form of a graph that relates relevant fragments to each other.

Fig. 1: Contrasting traditional problem diagnosis with our approach of context-based analytics.

application, that are *implicitly* connected. Deployment, runtime and development knowledge are required to establish the links between the fragments within this mental model.

In this paper, we propose an analytics approach that incorporates system and domain knowledge of runtime information to establish *explicit links* between fragments to build a *context graph*. Each node in the graph corresponds to a fragment (e.g., individual logs, metrics, or source-code excerpts). Edges correspond to semantic relations that link a fragment to related fragments, which we refer to as its *context*. Figure 1b illustrates context-based analytics as compared to the traditional approach in Figure 1a. It organizes the plethora of runtime information in a context graph where developers navigate the graph to inspect relevant connected fragments. The nature of graph nodes and its relations are defined in an initial modeling phase during which a context model is built. The graph is then constructed on-line from the modeled data sources.

We implemented a proof-of-concept prototype of the context-based analytics framework. Furthermore, we instantiated our prototype with an application-context model for a real cloud application at IBM. We applied our framework to diagnose two issues taken from the issue tracker of the studied application and found that our approach reduced the effort of diagnosing these issues. In particular, it decreased the number of required analysis steps by 48% and the number of needed inspected traces by 40% on average as compared to a standard diagnosis approach.

This paper makes the following contributions: (1) we described an approach to unify runtime traces and source code in a graph structure and to explicitly establish connections between the information fragments, which we refer to as *context-based analytics*; (2) we implemented the approach, provided a reference architecture description and open sourced the framework on Github; and (3) we conducted a case study with a real cloud application.

Next, we contrast our approach with related work (§ II), describe the framework for context-based analytics (§ III), elaborate on the framework implementation (§ IV), delve into a case study with a real cloud application (§ V), discuss the lessons we learned (§ VI) and limitations of our study (§ VII), and present some final remarks (§ VIII).

II. Related Work

Our work falls within the general topic of software analytics, and is particularly related to research on traceability and visualization of runtime behavior.

A. Software Analytics

There is a multitude of work that can be combined as pertaining to software analytics [24]. While most research in this area has investigated how to use static information (e.g., bug trackers and code repositories) to guide decisions, some studies rely on runtime traces to provide insights for software engineers. Often log analysis is used to understand system behavior [15], [22]. A recent study from Microsoft investigates how event and telemetry data is used by various roles in the organization [2]. These works usually focus on identifying one specific aspect of failing systems (e.g., performance [32], emergent issues [22]), whereas our approach constructs a graph for systematically exploring runtime information.

B. Traceability

There has been extensive research on traceability between requirements (and other textual artifacts) and source code.

Marcus and Maletic establish traceability links between documentation and source code using latent semantic indexing [23]. Spanoudakis et al. use a rule-based approach to infer these links [30]. In contrast, our work focuses on tracing various kinds of runtime information (including textual artifacts, such as log statements) to source code. The difference is mostly that runtime information (e.g., log statements) are much shorter than comparable software documentation and do not require such rigorous pre-processing as longer requirements documents. Linking runtime artifacts with code is probably more related to tracing in a performance modeling context [17].

C. *Visualization of Runtime Behavior*

Both systems and software engineering research have looked into different ways to understand runtime behavior through visualization. Sandoval et al. [28] investigate performance evolution blueprints to understand the impact of software evolution on performance. Bezemer et al. [4] investigate differential flame graphs to understand performance regressions. Cornelissen et al. [13] showed that trace visualizations in the IDE can significantly improve program comprehension. ExplorViz [14] provides live trace visualization in large software architectures. While existing work mostly focuses on a specific area of runtime information and one data source (e.g., performance as execution time from profilers or internal runtime behavior of objects from the JVM), we provide a framework to unify a diverse set of data from different data sources as a way to guide navigation on this search space.

D. *State of Practice.*

Industry tooling related to our approach mostly combines different metrics in dashboards. Probably the most prominent open-source tool chain in this area is the ELK stack[1] (Elastic-Search, Logstash, Kibana) where logs from distributed services are collected by Logstash, stored on ElasticSearch, and visualized in Kibana. More closely related to our approach is the open-source dashboard Grafana[2], that is mostly used to display time series for infrastructure and application metrics. Commercial counterparts to these services include, for instance, Splunk[3], Loggly[4], and DataDog[5]. The critique to the common dashboard solutions in current practice is that the amount of different, seemingly unrelated, graphs is overwhelming and it is hard to come to actionable insights [8]. Our approach attempts to give guidance to navigate the plethora of data by establishing explicit links between them. Further, to the best of our knowledge, DataDog is the only tool that attempts to correlate commits to other performance metrics. However, this correlation is based on temporal constraints of the commit only (similar to an idea in our own earlier work [10]) and does not involve any analysis of the code itself.

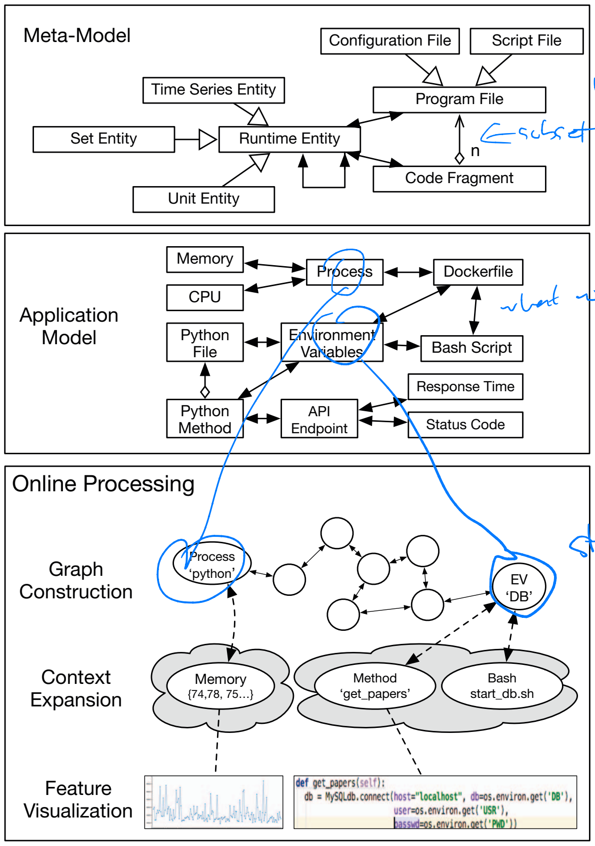
[1]https://www.elastic.co/webinars/introduction-elk-stack
[2]http://grafana.org/
[3]https://www.splunk.com/
[4]https://www.loggly.com/
[5]https://www.datadoghq.com

III. Conceptual Framework for Context-Based Analytics

Figure 2 provides an overview of the components and process of context-based analytics. Our approach makes previously implicit links between between runtime information and code fragments explicit by constructing a graph (*context graph*) based on an application context model. The application model is an instantiation of a meta-model, that defines the underlying abstract entities and their potential links of context graphs. The modeling process is only required as an initial step. With every problem diagnosis task, a graph is constructed with the current runtime state of an application. Selecting a node leads to its context being expanded that yields more nodes that are linked to the initial one. Finally, each node is visualized based on its feature type.

In the following, we describe the components of the framework in more detail.

This's how context-based analytics works



```
def get_papers(self):
    db = MySQLdb.connect(host="localhost", db=os.environ.get('DB'),
                         user=os.environ.get('USR'),
                         passwd=os.environ.get('PWD'))
```

Fig. 2: Overview of the context-based analytics approach from initial modeling to online processing of a context graph

A. Meta-Model

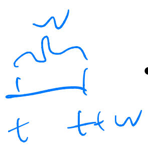
all entities are

and are linked by

We define a meta-model that defines abstract types and their possible relationships in the context graph that is depicted in Figure 2.

Runtime Entity is an abstract structure that includes any information that represents the state of an application and its underlying systems at a certain point in the time $t = \{1, \dots, T\}$. State is gathered either through observation (e.g., active monitoring [11] or system agents) or through log statements issued by the application. A runtime entity consists of a number of attributes $e_t = \{a_1, \dots, a_n\}$ that represent the application state at time t. In our meta-model, we differentiate between unit-, set-, and time series entities:

- *Unit Entity* refers to a single fact that is observed at time point t that does not belong to a group of observations (such as a set or a time series).
- *Set Entity* refers to a set of unordered observations at a time point t that exhibit common properties. An example of a set entity could be a set of processes running within a container at a certain point in time.
- *Time Series Entity* refers to a set of totally ordered observations starting at time t within the time window w. An example of a time series entity could be the evolution of execution times of a REST API endpoint within a time window.



A runtime entity can establish links to other kinds of runtime entities and *Program Files* and *Code Fragments*.

includes

Program File represents an abstract entity that encompasses all sorts of code files that relate to the application and are stored in version control (e.g., a Java file containing a class definition). A *code fragment* is a continuous set of code lines that is a subset of a program file.

We further differentiate between *configuration file* and *script file*. This distinction is modeled to enhance the basic capabilities of automatically establishing links.

- *Configuration File* refers to program files that set (initial) parameters for the application. This can also include infrastructure setup files (e.g., Dockerfiles). For a program file to qualify as a configuration file, it has to exhibit clear syntax and semantics towards providing configuration parameters (e.g., key-value pairs in .ini files, exposing ports in Dockerfiles).
- *Script File* refers to files that cannot be distinguished as either program files that yield application functionality, configuration, or operations task (e.g., batch processing).

For *1) Establishing Links:* Two entities are linked through a mapping $\mathcal{L} : \mathcal{E} \times \mathcal{E} \mapsto [0; 1]$, that represents the degree of connectedness between features as a normalized scale between 0 (not connected at all) and 1 (very high level of connectedness). Based on the abstract entities defined in the meta-model, our approach encompasses basic implementations of the function $\mathcal{L} : \mathcal{E} \times \mathcal{E} \mapsto [0; 1]$ to establish a link between two entities in $\mathcal{E}$[6]. On a high level, we follow the notion of semantic similarity in taxonomies [27] to describe relationships between entities. The more information two entities have in common, the more similar they are. For entities $e_i \in \mathcal{E}$ with attributes $a_1, \dots a_n$, this can be expressed as

most similar attributes among all attributes in two classes.

$$\mathcal{L}(e_1, e_2) = \max_{a_i, a_j \in S(e_1, e_2)} sim(a_i, a_j) \quad (1)$$

where $S(e_1, e_2)$ is a set of attributes contained in both e_1 and e_2 where a similarity function, sim, for their respective attribute type exists. In the following, we describe basic implementations to establish links.

a) Time Series to Time Series: Observations of metrics over time exhibit system behavior. Often a spike in a high-level metric is caused by one or several lower level components. This phenomenon is investigated by the means of correlating time series. Our approach incorporates basic time series correlation methods that can be parameterized in its correlation coefficient.

Example: Both CPU utilization and method response time are time series features of the system. Time series correlation analysis is applied to establish whether there might have been influence of CPU on response time. Note that, this does not necessarily establish a causal relationship.

b) Set/Unit to Code Fragment: In addition to establishing links between runtime traces, we also want to trace runtime information back to source code. To perform this matching, we distinguish between code fragments as an AST representation and code fragments as plain text:

- AST Representation: If we can parse the code fragments, we perform matching on the AST representation of the source code. For each node that contains text (variable names, method calls, strings literals, etc.) in the AST, we compare to attribute values of the set items or unit through string similarity.
- Plain Text Representation: In other cases, attribute values of the set items or unit are compared to the whole program fragment text through string similarity.

Example: An environment variable is used to configure aspects of a system. To properly assess the ramifications of their respective values in an application, a developer has to inspect the code in which the environment variable is used. We can link these features by matching the variable name in an AST node of a method or script.

For the remaining combinations of entity types, we attempt to map attributes of two features by attribute name. If entities $e_1, e_2 \in \mathcal{E}$ contain attributes with the same name, we apply string similarity on the attribute values of those who matched in name.

B. Application Model

contains

The application model is an instantiation of the meta-model. It describes all information required to construct the context graph for problem diagnosis. More specifically, the application needs to describe

[6]We use the term "entities" to describe both runtime entities and program files and code fragments

- Concrete Runtime Entities with a unique name and a set of attributes, and its type derived from the meta model
- Query to a data provider (e.g., SQL query, API call)
- Optionally, a method to extract attributes from an unstructured query result (i.e., feature extraction [18])
- Source Code Repository to extract program files
- Specification of links between entities
- If necessary, similarity measures for specific entity relationships

Given this information in the model, we can construct the context graph.

C. Context Graph

A context graph is a graph $G_t = \langle \mathcal{V}_t, E_t \rangle$ where nodes V_t describe the entities and edges E_t describe links. An edge between two nodes exists if the mapping $\mathcal{L}$ between the features yields a connectedness level above a certain threshold τ. All nodes connected through outgoing edges are called the *context* of the node.

1) Graph Construction: The aim of a context-graph is to support system comprehension and problem diagnosis by providing a representation of the problem space around application state and code fragments that can be *explored*. Due to the size of the problem space, manually inspecting and constructing the whole graph in a reasonable time is practically infeasible. Instead we opted for initially constructing the graph with a subset of “starting nodes", $V_S \in V$, that represent an entry point to the system. The starting nodes can be selected during the initial context modeling phase (as a default) or can be re-configured before the start of a diagnosis session. In the construction phase, the starting node values are retrieved from the data sources specified in the application model. Program fragment nodes are extracted from the code present in version control at the specified time t.

entities from the application model

Example: In Figure 2, we see an example of two starting nodes: The process 'python' and the environment variable 'DB'.

2) Context Expansion: To continue with the exploration of the problem space in the graph, we rely on a function that given an entity node $e_i \in \mathcal{E}$ in the context-graph can infer a set of context nodes

$$\Gamma(e_i) = \{ \ e_j \mid \mathcal{L}(e_i, e_j) > \tau \ \} \tag{2}$$

where τ is a pre-defined threshold to specify a minimum level of similarity for the features to qualify as connected.

The new nodes form a subgraph, the *context* of e_i,

the threshold

$$C(e_i) = \langle V(e_i), E(e_i) \rangle \tag{3}$$

of the initial graph G_t^w with

$$V(e_i) = \Gamma(e_i) \tag{4}$$

as nodes and

$$E(e_i) = \{ \ (e_i, e_j, v) \mid v = \mathcal{L}(e_i, e_j), v > \tau \ \} \tag{5}$$

as edges.

Example: Referring back to Figure 2, we see an illustration how context expansion works. By selecting the node *Process 'python'* the memory consumption in the service that is related to that process is linked. In a similar manner, when selecting the node for *Environment Variable 'DB'*, context expansion establishes a link with the code fragment that contains the method 'get_papers' and the Bash file 'start_db.sh'.

3) Entity Visualization: The framework provides standard visualization for the every component of the meta-model to support analysis. However, if an entity requires a specific visualization, the framework provides extension points to override the standard visualization.

IV. Implementation

We implemented a proof-of-concept of the context-based analytics framework with a backend in Python. The frontend works as a combination of the Jinja2[7] template engine and JavaScript with the d3 visualization library[8] and Cytoscape.js[9] to display and manipulate the context-graph. Figure 3 shows a screenshot of the implementation.

Basic Abstractions. The framework provides the abstract base entities described in the meta-model in Figure 2. It also provides abstractions for data queries to support modeling and inference of the context-graph. Currently, the framework provides data adapters for Elasticsearch and JSON (retrieved from the file system or over HTTP) for application state. It can retrieve code fragments over Git and attempts to provide an AST for code fragments with ANTLR [25]. So far, the implementation has only been tested with Python code of the case study application. However, it should work for any available grammar for ANTLR.

Extension Points. The implementation is architected to support extensibility. It provides the basic structures described in the conceptual framework in Section III with extension points to either extend or override the functionality for (1) entity-to-entity similarity measures, and (2) visualizations. For instance, given a very specific entity that is unique to an application (see “Adaptation" entity in RQ2 of Section V).

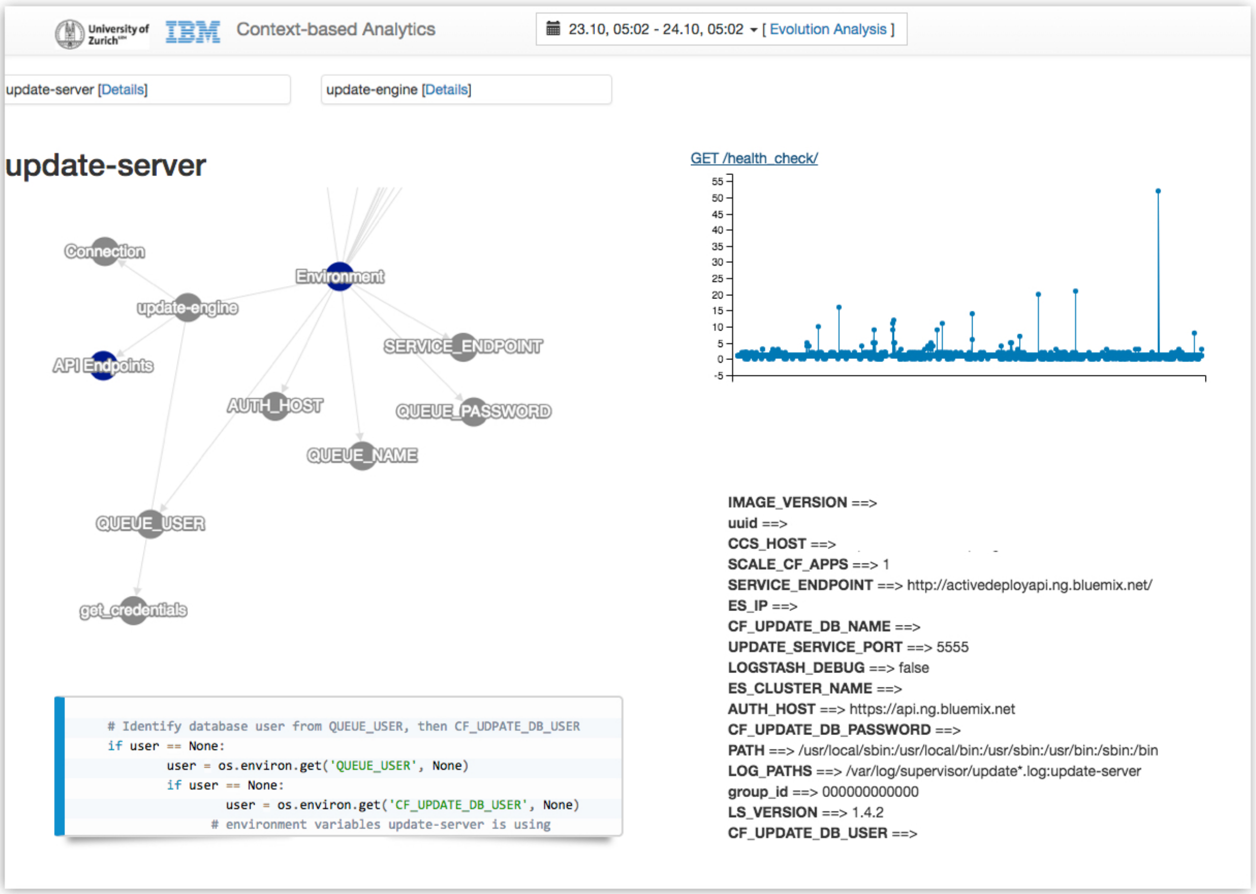
Components of the framework implementation are open-source and can be found online on GitHub[10].

V. Case Study

The goal of our case study was to evaluate to what extent the concept of context-based analytics can be used to diagnose runtime issues in real-life applications. Concretely, we investigate two research questions based on the proof-of-concept implementation discussed in Section IV.

1) RQ1 (Effort): How much effort (as measured in diagnosis steps taken and traces inspected) is required to diagnose a problem using context-based analytics as compared to a standard approach?

[7]http://jinja.pocoo.org/
[8]https://d3js.org/
[9]http://js.cytoscape.org/
[10]https://github.com/sealuzh/ContextBasedAnalytics



```
# Identify database user from QUEUE_USER, then CF_UPDATE_DB_USER
if user == None:
        user = os.environ.get('QUEUE_USER', None)
        if user == None:
                user = os.environ.get('CF_UPDATE_DB_USER', None)
            # environment variables update-server is using
```

```
IMAGE_VERSION ==>
uuid ==>
CCS_HOST ==>
SCALE_CF_APPS ==> 1
SERVICE_ENDPOINT ==> http://activedeployapi.ng.bluemix.net/
ES_IP ==>
CF_UPDATE_DB_NAME ==>
UPDATE_SERVICE_PORT ==> 5555
LOGSTASH_DEBUG ==> false
ES_CLUSTER_NAME ==>
AUTH_HOST ==> https://api.ng.bluemix.net
CF_UPDATE_DB_PASSWORD ==>
PATH ==> /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
LOG_PATHS ==> /var/log/supervisor/update*.log:update-server
group_id ==> 000000000000
LS_VERSION ==> 1.4.2
CF_UPDATE_DB_USER ==>
```

Fig. 3: Screenshot of the context-based analytics tool implementation. On the left the context graph and around it visualizations of a time series entity, set entity, and a code fragment. On the top right developers can set time t and window w

2) RQ2 (Generalizability): Can the context-based analytics approach also be applied to other case studies?

We have evaluated RQ1 is based on IBM's *active-deploy*[11] application, which is part of the IBM Bluemix Platform-as-a-Service (PaaS) environment. We have chosen two real-life issues that have occurred within *active-deploy* to measure diagnosis efforts. To address RQ2, we have conducted the *application modeling* step (see Figure 2) for a second case study application from the CloudWave European project [5].

RQ 1

A. RQ1 – Evaluation of Diagnosis Effort

1) Case Study Application: The case study application used to address RQ1 is IBM's *active-deploy*. *active-deploy* is a utility application that allows the release of a new cloud application version with no down time. It is implemented as a service-based architecture and consists of approximately 84 KLOC. Each individual service runs in its own Docker container. Runtime information is gathered in the form of system metrics and log messages. System metrics are observed through an agentless system crawler[12]. The metrics are available in a homegrown metric storage service over an API. Logs from the distributed services are collected through a log aggregator (Logstash) and stored on a central database (ElasticSearch).

2) Case Study Application Modeling: The framework implementation (see Section IV) provides basic components for (1) abstract entities in the meta-model, (2) interfaces to common data providers (e.g., JSON over an HTTP API), and (3) common visualizations for the entities in the meta-model (e.g., line chart for time series, syntax highlighting for code fragments). The application model is an instantiation of these basic components of the meta-model and defines entities with data sources and their links. For *active-deploy*, we depict the entities and the links of the application model in Figure 2. Every entity models a query through a data provider (e.g., SQL query, API call). We wrote a data provider for ElasticSearch and the IBM internal metric storage service. Further, we needed to provide the model with a `git` repository to extract code fragments.

[11]https://github.com/IBM-Bluemix/active-deploy

[12]https://developer.ibm.com/open/agentless-system-crawler/

In the following, we describe how we used this model to compare the effort of context-based analytics to a traditional baseline setting, in real-life problem scenarios.

3) Problem Scenarios: To study our approach based on realistic situations, we must evaluate them for runtime issues that are representative of common real-life situations. We are aware of two seminal works that discuss the underlying reasons of investigate runtime problems in terms of in Web- and cloud-based software. Firstly, Pretet and Narasimhan conclude that 80% of the failures are due to software failures and human errors [26]. Secondly, Hamilton has assume postulated that 80% of all runtime failures either originate in development, or should be fixed there [16]. Hence, we have sampled the issue tracker of *active-deploy* and have selected two example runtime problems (scenarios) with roots in software development that are considered "typical" by application experts from IBM. We have excluded problems that are purely operational, e.g., issues caused by hardware faults. In compliance with requirements from IBM with regards to not revealing the internal technical architecture of the case study application, we refer to the two relevant services within the case study application as *Service* 1 and *Service* 2.

we have service 1 & 2

Scenario 1 - Context: Environment Variable Configuration Mismatch. Service 1 and Service 2 relay messages to each other through a document database that acts as a message queue. Service 1 acts as the frontend to the application and writes tasks into the queue that Service 2 reads from. Service 2 (which is scaled horizontally to multiple cloud instances) writes heartbeat messages into the queue. Service 1 reads the heartbeat messages to determine the availability of Service 2 as the task executor. The engineers in the team receive an alert from operations indicating that *active-deploy* is down and need to investigate. we're gonna find The root cause turned out to be a mismatch of an environment variable in the code of Service 2, which led to the creation of a document database with the mistyped name.

Scenario 2 - Context: Service Performance Regression. Service 1 provides a REST API to allow cloud applications to be upgraded with zero downtime. The engineers in the team receive a call that interactions with cloud applications are significantly slower than usual. From the information available to the team it is not clear which specific REST API endpoint is affected. There have been no recent changes to find any of the endpoint's source code. After investigation, the root cause has been narrowed down to an additional expensive call added previously, which has only now started affecting the service performance due to changes in the service workload (higher usage).

We further divide those scenarios into 8 concrete subtasks (i.e., questions that an engineer diagnosing these problems needs to answer), which we have designed in collaboration with application experts. These subtasks are listed in Table I.

4) Methodology: In RQ1, we aim to establish the effort that an engineer has to go through to diagnose the two scenarios, including all 8 subtasks. We first design a company baseline case study as described by Kitchenham [19]. This baseline includes all data sources, tools, and workflows the

	Environment Variable Configuration Mismatch
T1	How long has the application been down?
T2	When did Service 1 last communicate to the queue?
T3	Is Service 2 alive and sends heartbeat messages to the database?
T4	Is Service 2 processing tasks from the database?
T5	How and where are the environment variables for the database set?
	Service Performance Regression
T6	When did the performance regression start?
T7	What endpoints are affected?
T8	What code has been changed on that endpoint?

TABLE I: Subtasks used in the diagnosis of two real runtime issues in *active-deploy*.

IBM team who developed the case study application used for problem diagnosis. We then compare our context-based analytics approach to this baseline using two metrics:

# of steps taken counts all distinguishable activities taken that might yield necessary information to be able to reach a conclusion of the given task. In the basline, steps include, but are not necessarily limited to:

- *Data Query*: Issuing a query to a data source (either within a log dashboard or directly in the database). This activity also includes refining and filtering data from a previous data query.
- *Data Visualization*: Plotting or visualizing data points.
- *Inspecting File Contents*: Opening a file to inspect it. This can be a file that is either in version control or on an operational system (e.g., log files).
- *Software History Analysis*: Inspecting the software history in version control (i.e., commit details) including their changes (e.g., changed files and diff's) [12].

Contrary, in context-based analytics, steps taken translates to expanding the context on a node in the graph (which in the background translates to many steps, that would have been taken manually in the baseline), or adjusting the time t or window w of observation.

# of traces inspected counts all information entities that have to be investigated to either solve the task or provide information that guide the next *steps to be taken*. Information entities in this context include, but are not limited to:

- *Log statements*: A log statement either in a file or within an aggregated dashboard (e.g., Kibana).
- *Graphical Representation of Data Points*: A plot/visualization of a time series, histogram, or similar, where the inspection of the graphical representation as a whole (as opposed to inspecting every single data point in the graph) leads to observing the required information.
- *Datastore Entries*: A row resulting from a query to a data store (including aggregation results).

The first author of the paper performed all subtasks for both, the company baseline and the context-based analytics tooling, and manually tracked the steps taken and traces inspected. The evaluation procedure has been validated by application experts. We provide a log of all actions taken during the case study and its detailed description in an online appendix[13]. For the baseline, all subtasks have been performed so that the number of steps taken and traces inspected are minimal based on the information provided by application experts. Hence, the following results represent a conservative lower bound of the benefits of context-based analytics.

5) Results: The shows results of this case study evaluation are summarized in Table II. Over all 8 tasks combined, our approach saved 48% of steps that needed to taken and 40% of traces that needed to be inspected. There is only a single subtask (T4) where using context-based analytics leads to a (slightly) increased number of traces that need to be inspected. For 3 subtasks (T3, T6, and T7), the number of inspected traces is unchanged. However, the number of steps taken increases substantially for all subtasks.

Context-base analysis

	Baseline		CBA		Effort Comparison	
Task	Steps	Traces	Steps	Traces	Δ Steps	Δ Traces
T1	4	2	2	1	-2 (50%)	-1 (50%)
T2	2	3	1	2	-1 (50%)	-1 (33%)
T3	2	2	1	2	-1 (50%)	0 (0%)
T4	2	7	1	8	-1 (50%)	+1 (-12.5%)
T5	5	42	3	18	-2 (40%)	-24 (43%)
T6	5	2	2	2	-3 (60%)	0 (0%)
T7	2	18	1	18	-1 (50%)	0 (0%)
T8	3	17	2	5	-1 (33%)	-12 (70.5%)
Total	**25**	**93**	**13**	**56**	**-12 (48%)**	**-37 (40%)**

TABLE II: Case study results for RQ1. Using context-based analytics, the number of diagnosis steps that need to be executed are reduced by 48%, and the number of traces that need to be inspected are reduced by 40%. These numbers represent improvements over an idealized baseline, where engineers do not "waste time" with unnecessary diagnosis steps or traces that are not relevant to the issue at hand.

Note that these numbers represent an idealized situation and do not contain any unnecessary diagnosis steps or traces for either the baseline or our approach (i.e., the evaluation assumes that developers never run into a dead end while diagnosing the issue). Further be aware that a "step" in both approaches constitutes different things. In the baseline setting, a step can be a complex query to a database, fine-tuned to retrieve specific results. Conversely, in our approach, a "step" is only expanding a context node in the prototype implementation, or setting a different time t or window for time series data w. Further, specific domain knowledge is often necessary to construct proper queries to available data sources. Through our approach, this domain knowledge is also required, but encoded in the initially constructed application context model. All in all, we expect the real-life effort savings of using context-based analytics to even be underrepresented by the above numbers, especially for a novice engineer, or an engineer who is not an expert of the application that should be diagnosed.

RQ 2

B. RQ2 – Evaluation of Generalizability

To get an idea whether the meta-model of context-based analytics can be generalized, we conduct the application modeling step for another industrial application.

1) Case Study Application: The case study is a larger application from a major telecommunications provider that is deployed within the CloudWave European project [5]. It was chosen as a representative instance of a cloud application with distributed logs tracking multiple scalable components. The project is implemented as a service-based architecture and consists of approximately 139 KLOC. It is deployed in an OpenStack environment[14]. Runtime information are gathered in the form of system metrics, log messages, and application metrics. System metrics are mostly gathered through the underlying platform, OpenStack and stored in the service Ceilometer[15], that acts as a "metric hub". Application metrics are sent directly from the application through a low-level system daemon (similar to StatsD[16]) where they are stored in a document database (MongoDB). Further, application metrics are also derived from low-level metrics that are aggregated through complex-event-processing in Esper [21]. These are then relayed through a homegrown tool called CeiloEsper[17].

2) Case Study Application Modeling: We modeled this additional case study by retrieving a list of data sources and its metrics as *metric name, metric type, metric unit*. We established the links between entities based on our understanding of the application and discussed the results with application experts and operators of the CloudWave project. The entities and links of the resulting application model are depicted in Figure 4. this's

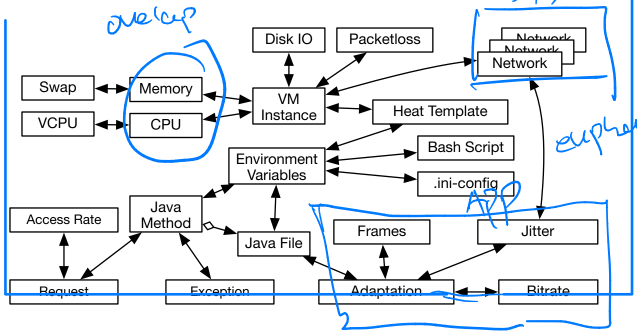


Fig. 4: Application model of an application in the telecommunication industry deployed in OpenStack from the CloudWave European project

3) Comparison: The model has some overlap with the model of our first case study at IBM, mostly regarding lower level machine metrics (e.g., CPU, Memory). Being an application in the telecommunications industry, it has an emphasis on infrastructure and network metrics both on systems level (the "Network" entities on the top right in Figure 4 include incoming and outgoing bytes/packets/datagrams, read and write rates of bytes, active connections, etc.) and application level (e.g., jitter, average bitrate, number of frames). *Adaptations* are im-

[13]Online Appendix: https://sealuzh.github.io/ContextBasedAnalytics/

[14]https://www.openstack.org/

[15]https://github.com/openstack/ceilometer

[16]https://github.com/etsy/statsd

[17]https://github.com/MDSLab/cloudwave-ceilometer-dispatcher

portant events in the domain of the application [6] (application-specific functionality degradation). Diagnosing why specific adaptations have happened in CloudWave is well supported by context-based analytics, as adaptations are triggered by the application metrics *Jitter*, *Bitrate*, and *Frames*. We can also establish a link back to the Java code file, since the adaptation actuators are marked in the code through Java annotations.

The abstractions in the meta-model proved flexible enough to appropriately model both case study applications from vastly different domains. However, the visualizations provided by the current proof-of-concept implementation built for IBM are limited in their expressiveness, and not generally useful to the CloudWave implementation without changes. For instance, *Adaptation* in this case study is a time-series entity, but a line chart of simply a boolean value (adaptation happened or not) over time can be improved. However, This the implementation Is solved counters this by offering extension points to assign different visualizations to specific entities.

VI. Discussion

Based on our case study implementation and discussions with practitioners, we have identified a number of interesting challenges and discussion items in the scope of context-based analytics.

The need for dynamic context models. In the current state, the application context model is designed once as part of an initial modeling phase, and then instantiated throughout the use of our approach. This is especially favorable for junior engineers and newcomers to the project, as it allows them to systematically browse traces with little to none of the required domain knowledge. However, in the course of our case study, it became apparent that allowing dynamic queries to the existing data repositories [31] to extend the context model on the fly can be beneficial for more senior engineers. Otherwise, our approach faces the danger of being perceived as a rigid corset that limits experts rather than supporting them. Hence, as part of future work, we will investigate an extension to the context-based analytics meta-model to support the creation of a dynamic, ad-hoc context model.

Integrating business metrics. Our framework conceptually allows for the integration of various kinds of runtime data. However, so far, we have exclusively focused on the aggregation of technical runtime data (e.g., response times, CPU utilization) to static information, such as code changes. An interesting extension of this idea is to also integrate metrics that are more related to the business value of the application, for instance click streams, conversion rates, or the number of sign-ups. We expect that our meta-model is already suitable to allow for integrating business metrics. However, whether other extensions are practically necessary is an item for future investigations.

Data privacy. To construct the context graph we solely use information that is already available through existing channels in the organization. We only establish explicit links that beforehand were implicit, and, thus, probably harder to retrieve. However, automated event correlation and data aggregation can still be the cause of privacy concerns. In previous work, we argued for collaborative feedback filtering as part of organization-wide governance [9]. Collaborative, privacy preserving data aggregation has been discussed as a solution to this problem in the past [1], [7].

VII. Threats to Validity

We now discuss the main threats to the validity of our case study results.

Construct Validity. We have constructed our evaluation regarding RQ1 and RQ2 carefully following the established framework introduced by Kitchenham [19]. However, there are still two threats that readers need to take into account. Firstly, we have chosen *# of steps taken* and *# of traces inspected* as observed metrics in RQ1 to represent the effort that engineers have to go through when diagnosing runtime issues. These metrics are necessarily simplified to allow for objective comparison. However, in practice not all steps and traces are the same, e.g., some diagnosis steps may be more difficult and time-consuming than others. Secondly, there is the threat that we have overfitted our general context-based analytics framework or prototype implementation for the specific *active-deploy* use case. We have mitigated this threat by additionally applying our approach to a second use case as part of our evaluation of RQ2.

Internal Validity. For RQ1, the first author manually retrieved the observed metrics, which is subject to evaluator bias. We mitigated this threat by documenting the process in detail. Further, the second author, who is also an application expert of the case study application, has independently validated the diagnosis steps of the baseline setting. Additionally, we are as explicit as possible in describing how the tasks are being evaluated, and provide the documented process as an online appendix.

External Validity. A fundamental question underlying all case study research is to what extend the results generalize to other comparable applications (e.g., multi-service cloud applications outside of IBM). This threat was the main reason for our investigation of RQ2. While we were unable to report on the second case study in the same level of detail as for the first case due to data confidentiality issues, we were able to develop a suitable application model following the context-based analytics meta-model also for this second independent case, without requiring changes to the meta-model. Another threat to the external validity of our results is the question whether the two real-life issues selected in our evaluation of RQ1 are representative of common problems. We have mitigated this threat by carefully selecting real issues in collaboration with application experts, and based on suggestions from earlier literature.

VIII. Conclusion

In this paper, we have addressed the problem of diagnosing runtime problems in large-scale software. We have illustrated that problems that are observed in production, such as outages or performance regressions, often originate in development-time bugs. However, actually identifying the problematic changes in program or configuration code requires inspecting multiple fragments of disperse information that are implicitly connected. Currently, multiple different tools need to be used in conjunction (e.g., log aggregation tools, source code management tools, or runtime metric dashboards), each requiring different data sources and workflows.

We proposed context-based analytics as an approach to make these links explicit. Our approach consists of a meta-model, which needs to be instantiated in an initial modeling phase for a concrete application, and an online processing phase, in which the context model is supplied with concrete runtime data. We instantiate our context model for two real cloud computing applications, one from IBM's Bluemix PaaS system, and one from a major telecommunications provider. Using two concrete practical runtime problems from the Bluemix-based application's issue tracker, we show that using context-based analytics we are able to reduce the number of analysis steps required by 48% and the number of inspected runtime traces by 40% on average as compared to the standard diagnosis approach currently used by application experts. Our future work will include investigating the dynamic creation of context models to support ad-hoc querying, the integration of business metrics in addition to technical runtime metrics, and the investigation of privacy concerns in our approach.

The model is also tested for its generalizibility in an industrial application. Overall, the model proved its flexibility on modelling but unuseful visolizations.

References

[1] B. Applebaum, H. Ringberg, M. J. Freedman, M. Caesar, and J. Rexford. Collaborative, privacy-preserving data aggregation at scale. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 56–74. Springer, 2010.

[2] T. Barik, R. DeLine, S. Drucker, and D. Fisher. The bones of the system: a case study of logging and telemetry at microsoft. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 92–101. ACM, 2016.

[3] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[4] C.-P. Bezemer, J. Pouwelse, and B. Gregg. Understanding software performance regressions using differential flame graphs. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 535–539. IEEE, 2015.

[5] D. Bruneo et al. Cloudwave: where adaptive cloud management meets devops. In *Proceedings of the Fourth International Workshop on Management of Cloud Systems (MoCS 2014)*, 2014.

[6] D. Bruneo, F. Longo, and B. Moltchanov. Multi-level adaptations in a cloudwave infrastructure: A telco use case. In *Advances in Service-Oriented and Cloud Computing - Workshops of ESOCC 2015*, 2015.

[7] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. *Network*, 1:101101, 2010.

[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 2015, pages 393–403, New York, NY, USA, 2015. ACM.

[9] J. Cito, P. Leitner, H. C. Gall, A. Dadashi, A. Keller, and A. Roth. Runtime metric meets developer: building better cloud applications using feedback. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 14–27. ACM, 2015.

[10] J. Cito, G. Mazlami, and P. Leitner. Temperf: Temporal correlation between performance metrics and source code. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*, QUDOS 2016, pages 46–47, New York, NY, USA, 2016. ACM.

[11] J. Cito, D. Suljoti, P. Leitner, and S. Dustdar. Identifying root causes of web performance degradation using changepoint analysis. In *International Conference on Web Engineering*, pages 181–199. Springer, 2014.

[12] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey. Software history under the lens: a study on why and how developers examine it. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 1–10. IEEE, 2015.

[13] B. Cornelissen, A. Zaidman, and A. Van Deursen. A controlled experiment for program comprehension through trace visualization. *Software Engineering, IEEE Transactions on*, 37(3):341–355, 2011.

[14] F. Fittkau, S. Roth, and W. Hasselbring. Explorviz: Visual runtime behavior analysis of enterprise application landscapes. AIS, 2015.

[15] Q. Fu, J.-G. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie. Contextual analysis of program logs for understanding system behaviors. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 397–400. IEEE Press, 2013.

[16] 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.

[17] R. Heinrich. Architectural run-time models for performance and privacy analysis in dynamic cloud applications. *ACM SIGMETRICS Performance Evaluation Review*, 43(4):13–22, 2016.

[18] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):249–267, 2008.

[19] B. A. Kitchenham. Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *ACM SIGSOFT Software Engineering Notes*, 21(1):11–14, 1996.

[20] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 185–194. ACM, 2010.

[21] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar. Application-level performance monitoring of cloud services based on the complex event processing paradigm. In *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8. IEEE, 2012.

[22] Q. Lin, J.-G. Lou, H. Zhang, and D. Zhang. idice: problem identification for emerging issues. In *Proceedings of the 38th International Conference on Software Engineering*, pages 214–224. ACM, 2016.

[23] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135. IEEE, 2003.

[24] T. Menzies and T. Zimmermann. Software analytics: so what? *IEEE Software*, 30(4):31–37, 2013.

[25] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[26] S. Pertet and P. Narasimhan. Causes of failure in web applications (cmu-pdl-05-109). *Parallel Data Laboratory*, page 48, 2005.

[27] P. Resnik. Using information content to evaluate semantic similarity in a taxonomy. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 448–453, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[28] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker. Performance evolution blueprint: Understanding the impact of software evolution on performance. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–9. IEEE, 2013.

[29] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. C. Gall. An Empirical Study on Principles and Practices of Continuous Delivery and Deployment. *PeerJ Preprints 4:e1889v1*, 2016.

[30] G. Spanoudakis, A. Zisman, E. Pérez-Minana, and P. Krause. Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72(2):105–127, 2004.

[31] C. Sun, H. Zhang, J.-G. Lou, H. Zhang, Q. Wang, D. Zhang, and S.-C. Khoo. Querying sequential software engineering data. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 700–710. ACM, 2014.

[32] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, 2014.