REVELIO: ML-GENERATED DEBUGGING QUERIES FOR FINDING ROOT CAUSES IN DISTRIBUTED SYSTEMS

Pradeep Dogga [1] **Karthik Narasimhan** [2] **Anirudh Sivaraman** [3] **Shiv Kumar Saini** [4] **George Varghese** [1] **Ravi Netravali** [2]

ABSTRACT

A major difficulty in debugging distributed systems lies in *manually* determining which of the many available debugging tools to use and how to query that tool's logs. Our own study of a production debugging workflow confirms the magnitude of this burden. This paper explores whether a deep neural network trained on past bug reports and debugging logs can assist developers in distributed systems debugging. We present Revelio, a debugging assistant which takes user reports and system logs as input, and outputs debugging queries that developers can use to find a bug's root cause. The key challenges lie in (1) combining inputs of different types (e.g., natural language reports and quantitative logs) and (2) generalizing to unseen faults. Revelio addresses these by employing deep neural networks to uniformly embed diverse input sources and potential queries into a high-dimensional vector space. In addition, it exploits observations from production systems to factorize query generation into two computationally and statistically simpler learning tasks. To evaluate Revelio, we built a testbed with multiple distributed applications and debugging tools. By injecting faults and training on logs and reports from 800 Mechanical Turkers, we show that Revelio includes the most helpful query in its predicted list of top-3 relevant queries 96% of the time. Our developer study confirms the utility of Revelio.

1 INTRODUCTION

Developers often must translate informal reports about problems provided by a user into actionable information that identifies the root cause of a bug. An ever-growing list of debugging tools aid developers in such root cause diagnosis. These tools can log the behavior of applications running on end hosts (Appdynamics.com, 2022; Opentracing.io, 2022; GNU.org, 2022), end host networking stacks (Tcpdump.org, 2022; McCanne & Jacobson, 1993), and network infrastructure (Narayana et al., 2017; INT, 2021). Some tools also track and relate execution across subsystems in a distributed system (Mace et al., 2015; Sigelman et al., 2010; Zipkin.io, 2022). In addition, each tool allows developers to query the collected logs (e.g., using BPF expressions or SQL queries) to hone in on interesting data.

Yet, debugging distributed systems remains difficult, largely because it typically involves multiple *manual* steps (Abuzaid et al., 2018): understand user reports[1], iteratively issue debugging queries to test hypotheses about potential root causes, and finally develop a fix. As a concrete example, consider debugging a client interaction with a web service that results in an unusually slow page load. The developer receives a user report about the incident and has to develop a bug fix. However, the problem could be in many possible subsystems. At the client, the browser could be executing malformed HTML. At the server, there could be an unreachable database, a program error in the application frontend, or a misconfigured network forwarding table.

In the example above, the developer's difficulty is not the lack of tools; on the contrary, dozens of rich debugging frameworks exist for each subsystem (§7). Instead, the challenge lies in manually determining which debugging queries to issue, on which subsystems' logs, and with what parameters. Further, the answer to each question above can depend on vast and heterogeneous logs. Indeed, distributed systems increasingly comprise many loosely coupled subsystems or microservices (Lyft Engineering, 2022), each with their own logging framework(s). As a result, developers face a significant cognitive burden to understand and correlate debugging information that exhibits heterogeneity in (1) data types (e.g., natural language reports, numerical switch counters), (2) data sources (e.g., network infrastructure, end-host stack), and (3) abstraction levels (e.g., user reports, system logs, network counters).

[1]University of California, Los Angeles [2]Princeton University [3]New York University [4]Adobe Research, India. Correspondence to: Pradeep Dogga <dogga@cs.ucla.edu>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2022.

[1]We focus on user reports, but note that our approach generalizes to auto-generated natural language crash reports.

Today, developers overcome these challenges using their hard-won intuition from debugging similar problems in the past, remembering which subsystems they investigated, and what debugging queries they issued. However, our developer survey and analysis of 4 months of debugging reports at a major SaaS company (*Anon*) revealed that this manual approach consumes significant developer time (§2). At *Anon*, moving from a user report to a root cause took developers 8.5 hours on average, despite the use of state-of-the-art debugging tools, and the fact that 94% of faults were repeat instances of the same few types (e.g., resource underprovisioning), with only locations varying. Prior studies of other production systems have similarly noted the significant time spent on root cause analysis, relative to tasks such as triaging (Chen et al., 2019; Pham et al., 2020).

In this paper, motivated by the prevalence of large historical debugging datasets in software organizations (Jackson, 2021; Orate, 2019) and the recurring nature of faults, we ask: *can a machine-learning (ML) model learn developer intuition from past debugging experiences to accelerate root cause finding?* More precisely, given debugging data from when a report was provoked, can an ML model automatically generate a debugging query that allows the developer to extract the most informative subset of logs? Further, is the model's output useful: does it let the developer diagnose bugs faster than the status quo?

To answer these questions, we developed **Revelio** (Figure 1), whose goal is to help developers use existing debugging tools more effectively. Revelio takes as inputs user reports and system logs from the target deployment, and outputs a ranked sequence of debugging queries that (when executed) elucidate the root cause. We chose to output a query sequence for several reasons. First, as evidenced by our production debugging analysis (§2), much developer time and effort in root cause analysis is spent selecting a subsystem to investigate, and determining how to use existing querying (or dashboard) tools to analyze its collected logs; queries capture both aspects. Second, bugs can often be tackled from multiple vantage points in distributed systems, e.g., congestion between two microservices can be resolved by either moving an application VM or changing inter-service routing rules. A sequence ensures that developers can extract root cause insights from those different vantage points to help determine the appropriate fix.

1.1 Challenges and Contributions

1. Extensible model using distributed vector representations. To combine *heterogeneous data sources* (e.g., natural language user reports, numerical switch counters), we use neural networks that map each input to a high-dimensional distributed vector representation (Alon et al., 2019; Mikolov et al., 2013), akin to intermediate representations like SSA (Appel, 1998) in programming languages.

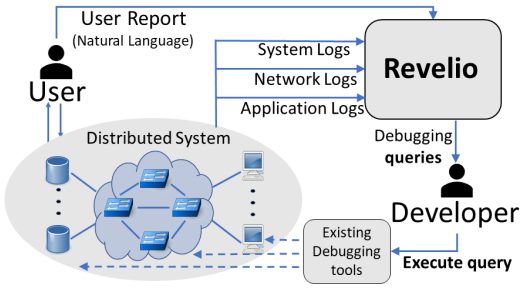

Figure 1: **Revelio takes as input a user report and system logs, and outputs (for the developer) a ranked sequence of debugging queries that highlight the root cause.**

This makes our architecture *extensible*: a new type of debugging data can be incorporated by learning a mapping from that data type to a high-dimensional vector.

2. Modeling queries as vectors using graph convolutional networks (GCNs). Ideally, we should be able to convert queries into the same vector representation as our inputs; we could then find the relevance of a query to a particular debugging scenario by applying standard machine-learning concepts such as a similarity score between the query and input vectors. One approach is to simply assign a unique label to each query and employ a multi-label classifier to generate queries. However, this performs poorly (§6.2) because the opacity and independence of such labels fail to exploit the fact that debugging queries for a tool are drawn from the same grammar. Instead, debugging queries are more faithfully modeled as abstract syntax trees (ASTs) in the syntax of the query language. To leverage this richer query format, we use GCNs (Kipf & Welling, 2016) to convert query ASTs into the same vector representation as our inputs.

3. Handling a large search space of queries using modularization. The search space of potential queries is massive due to the presence of many (1) *query templates*, i.e., skeleton queries for a given subsystem with unspecified parameters, and (2) *query parameters* that cover the scale of production systems (e.g., every IP address in the system could be a candidate value for a parameter). To handle this large search space, we *modularize* our ML model into two components. The first model uses user reports and system logs to *predict a query template*; then the second model uses only the predicted template and system logs (not user reports), to *predict numeric parameters*. This is motivated by our finding that production faults typically involve recurring types (§2), and can thus be debugged using a small set of templates–one per fault type. Modularization shrinks the output space of the first model, simplifying training computationally, and the input space of the second model, making it less likely to overfit to spurious features.

4. Generalizing to unseen faults using abstraction. To handle a large fraction of bugs in production settings, an ideal model should generalize to output useful queries for

occurrences of previously seen bugs at new locations (§2). To do this, we transform concrete switch/function ids into new, *abstract* ids based on rank on some metric, e.g., queue size; consequently, the ids in one setup and another need not be the same, allowing us to generalize to new fault locations. For example, if the model captures a dependence on the largest router queue (which in the training set was Router X), it can generalize during testing to a different Router Y with the largest router queue.

1.2 Evaluating Revelio

New distributed systems debugging testbed. While organizations running distributed systems routinely collect the described debugging data, much of it is proprietary. Thus, to evaluate Revelio, we built a testbed (§5) on top of the Mininet emulation platform (Lantz et al., 2010). Our testbed currently integrates four debugging tools: Jaeger (Jaegertracing.io, 2022), Marple (Narayana et al., 2017), cAdvisor (Google, 2022), and tcpdump (Tcpdump.org, 2022). It supports three industry-developed distributed applications: Reddit (Reddit, 2016) (monolithic), Sock Shop (Weaveworks, 2017) (microservice), and Online Boutique (Google, 2019) (microservice). In addition, our testbed includes an automatic fault injector that was informed by our analysis of production bugs (§2 and §5.3). To generate a realistic debugging dataset, we enlisted 800 Mechanical Turk users to interact with 85 faulty versions of each application and paired the uploaded user reports with system logs from the testbed. The testbed, dataset, and Revelio are available at `https://github.com/debugging-assistant/`.

Testbed evaluation, developer study. We evaluated Revelio using the above dataset, and with a developer study (§6.3). Our key findings are: **1.** across the set of potential queries supported by our debugging tools, for repeat occurrences of the same faults, Revelio ranks the correct (i.e., the one that most directly highlights the root cause) query in the top-k 96% (k=3), 100% (k=4), and 100% (k=5) of the time, **2.** Revelio's model successfully generalizes to output the correct query 87% (k=3), 88% (k=4), and 100% (k=5) of the time for faults that manifest in previously unseen locations, and **3.** developers with access to Revelio correctly identified 90% of the root causes (compared to 60% without Revelio), and did so 72% (14 mins) faster. We additionally conducted quantitative experiments that demonstrate the importance of each of our design choices (i.e., abstraction, GCNs, and modularization) and show that Revelio outperforms simpler ML approaches.

2 Root Cause Analysis in Production

To understand the operation and limitations of debugging tools and workflows in production distributed systems, we conducted a study at a major SaaS company (*Anon*). Our analysis involved 7 services at *Anon* that collectively handle 83 million user requests per day. Across these services, we examined the debugging process through a developer survey and a manual analysis of completed debugging tickets over a 4-month time period. Our analysis and results follow a general taxonomy based on a literature survey we conducted of publicly reported bugs in production distributed systems (§A.1).

Debugging workflow. Developers at *Anon* use a variety of state-of-the-art monitoring tools (e.g., Splunk (Splunk.com, 2022), Datadog (Datadoghq.com, 2021), others (Lightstep.com, 2022; Newrelic.com, 2022; Pingdom.com, 2022; Icinga.com, 2022)) that continuously analyze system logs, visualize that data with dashboards, and raise alerts when anomalous or potentially buggy behavior is detected. Alerts are raised on a given time-series based on either manually-specified heuristics and thresholds, or standard statistical analysis techniques that compare recent data to historical baselines (Lightstep.com, 2022; Taylor & Letham, 2018; Xu et al., 2018; Twitter Engineering, 2015). As user or internal reports are filed, the burden of debugging falls largely to developers.

For each report, developers must (1) filter through the raised alerts (across subsystems) to determine which are worth investigating and pertain to actual bugs and the issue at hand (vs. false positives), and (2) for bugs, find the root cause by analyzing the system as a whole. Both steps involve iteratively analyzing low-level system logs, inspecting prior debugging tickets and the current report (both written in natural language), and issuing debugging queries using interfaces that run atop the same logs used to raise alerts (Grafana.com, 2022; Narayana et al., 2017). Once a root cause is identified, a summary of the issue, root cause, and debugging process (e.g., investigated subsystems, issued queries) is documented as a completed ticket.

Analysis of historical debugging tickets. We manually analyzed all 176 debugging tickets that were created for the aforementioned services between November 2019 and February 2020. Our analysis involved manually clustering the tickets according to their root causes as documented by *Anon*'s developers. Table 1 summarizes our findings, from which we make three primary observations:

1. A few recurring categories of root causes collectively represent the vast majority (94%) of bugs.
2. The faults in a given category often manifest at different locations in the distributed system. For example, numerous "Resource underprovisioning" tickets involve high CPU loads but pertain to different servers, e.g., gateway vs. storage servers.
3. Identifying the root cause for a fault is time consuming, taking an average of 8.5 hours (min: 14 min, max: 2.9 days). We found that these lengthy durations are

Table 1: **Summary of closed debugging tickets at *Anon* over a 4-month period. Examples have been partially anonymized. Debugging times are reported in minutes.**

Root-Cause Category	# Tickets	# Locations	Example Root Cause	Avg. Diagnosis Time
Resource under-provisioning	17	11	Load balancer is consuming all available memory and starving other co-located services	293
Component failures	58	29	3 nodes for a service were down, leading to queued 400 ERRORs	176
Subsystem misconfigs	11	7	Incorrect host mapping configuration in Zookeeper caused a failure, and prevented cluster from servicing any events	276
Network congestion	5	4	A spike in wide-area traffic caused unusually low data transfer rates between city1 and city2	725
Network-level misconfigs	18	10	Instances in a region are pointing to a NAT instance with incorrectly configured security groups, leading to dropped traffic	92
Subsystem/Source-code bugs	31	22	Service returning 5xx errors due to a code change that added a condition on the availability of a parent asset ID	1607
Incorrect data exchange	26	16	4xx errors were being raised because the noise classifier service is sending additional data with each stock request	417
One-off/unknown	10	8	278 customer accounts were canceled for unknown reason	464

largely a result of the error-prone nature of root cause analysis: developers at *Anon* must explore multiple subsystems (5 on average) and issue many debugging queries (8 on average) to find the root cause of a fault.

Takeaways. Our findings at *Anon* collectively show that, while debugging tools have considerably improved (mainly through improved alert-raising and richer query interfaces), post-alert debugging, or moving from alerts to root causes of faults, is *largely manual* and *time consuming*. These difficulties show that existing monitoring tools and anomaly detectors commonly fail to elucidate the root cause of the problem, and still require much developer effort to synergize past and present system-wide data. The focus of this paper is on automating the post-alert process for developers, i.e., ingesting diverse system logs *and* natural language reports to automatically suggest debugging queries.

Problem tackled by Revelio. Query-driven debugging is already a part of the developer ecosystem, and queries integrate directly with other common tools, e.g., pairing queries with dashboards that display logs of interest. Revelio outputs debugging queries that highlight the root cause of a bug. To the best of our knowledge, there does not exist a solution for automating such query generation. Further, we believe that ML is the appropriate tool as it is unclear how to design automated heuristics that incorporate such diverse data sources and output full-fledged debugging queries (as opposed to, say, scalar alert thresholds). Additionally, the repetitive nature of faults also makes debugging amenable to an ML approach. Revelio is intended to augment (not replace) developers in the debugging process for two reasons. First, there often exists multiple ways to address a problem, and the 'right' one may depend on context beyond the purview of Revelio, e.g., financial constraints. A sequence of debugging queries can help characterize the root cause from multiple vantage points in the system. Outputting queries rather than a final fix provides a failsafe to ensure that suboptimal or incorrect fixes do not end up in production services.

Goals and non-goals. We note that our focus here is entirely on faults that fall into recurring categories—recall that such faults constitute the vast majority of faults at *Anon*, and despite their recurring nature, still take significant time to diagnose. Importantly, we *do not* target new fault categories and one-off faults that bear no similarity to prior ones, and instead leave debugging of those scenarios to future work. Additionally, our goal is to assist in the debugging process *after* a user report or an automatic alert has been detected. Thus, Revelio is not intended to replace statistical anomaly detectors, but instead to help with root cause finding *after* the anomaly detector has raised an alert.

3 Overview of Revelio

At a high level, Revelio takes two inputs: (1) a *user report* filed by a system user, and (2) the *system logs* collected during the user's interactions with the system. The two sources provide distinct perspectives into the state of the system when a fault occurs: the former from an external and the latter from an internal viewpoint. Further, the two data sources differ fundamentally: system logs are highly structured, accurate, and contextually close to a developer's debugging options; user inputs are often noisy, unstructured (e.g., raw text), and abstract with respect to low-level execution (e.g., a user may report that the system is slow to respond with no further information). As its output, Revelio generates a ranked list of top-k debugging queries that are directly executable on the target debugging framework(s) and highlight the root cause of the fault.

3.1 Challenges

Revelio must overcome four key challenges to generate debugging queries. First, the model has to combine and re-

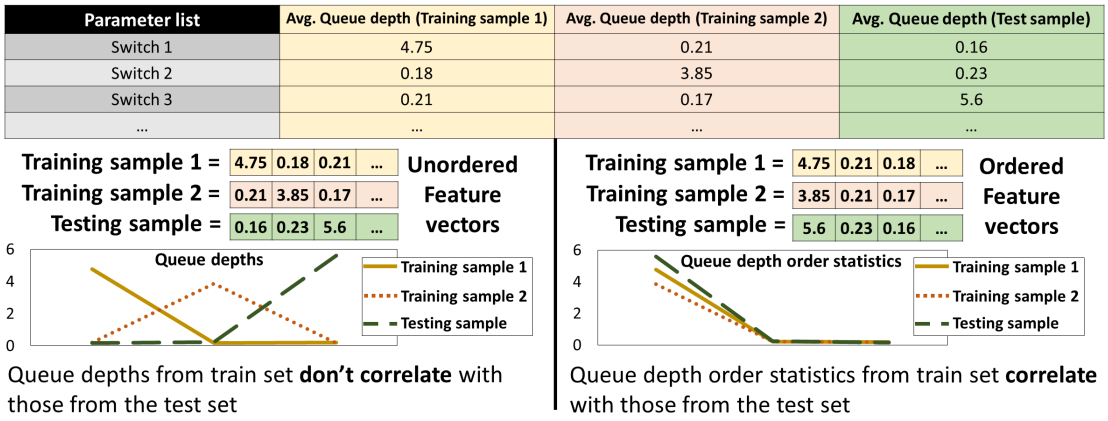

Figure 2: **Example showing how rank-ordering helps to generalize to faults of the same type at different locations. After ordering (right), despite the fault location being different, the queue depth order statistics in testing are correlated with those in training. In contrast, without ordering (left), the unseen fault location results in queue depth values that are dissimilar from training data.**

late diverse and seemingly disparate data inputs. Second, the output space of queries is highly structured, making it harder than standard multi-label classification where each label is independent (BakIr et al., 2007). This is because all debugging queries for a tool are drawn from the same language grammar, unlike opaque and independent labels. Third, the space of potential queries for a given input is large, requiring new techniques to scale to large distributed systems. Fourth, as per §2, the model must generalize in a specific sense: if a fault occurs at one location during training and is debugged with a specific query, then, during testing, the model must predict the same query with a different parameter if the same fault occurs at a different location.

3.2 Solutions

Challenge 1: Diverse data. We handle diverse data sources by converting each into a vector and concatenating all vectors to form the system state vector. This has two benefits. First, each data source is normalized for downstream operations in the ML model. Second, the architecture is extensible: a new data source (e.g., crash reports) can be added by converting it into a vector (either learned or manually) that is then concatenated with the existing system state vector.

Challenge 2: Predicting queries. To generate queries, which can be represented as abstract syntax trees (ASTs) in the grammar of a tool's query language, we employ a Graph Convolutional Network to convert the AST into a query vector. A vector-based representation is easier to use with the rest of the ML model relative to richer representations such as trees. During training, given pairs of query and system vectors, we find model parameters that maximize the probability that these query vectors were predicted from these system vectors. During inference, given the ML model's parameters, we find the query that maximizes the probability of a query vector given the system vector.

Challenge 3: Scaling to large systems. Revelio has to search over a large space of queries to output the best query in response to a given input. This search space scales with the size of the distributed system. To handle this, we exploit modularity and factorize our ML model into two cascaded components. The first uses user reports and system logs to generate query templates, which are skeleton queries for a particular subsystem with all numeric parameters left unspecified (e.g., SELECT _ FROM _). The second component then predicts the corresponding parameters using only the predicted template and system logs.

This approach is motivated by two ideas. First, production faults typically involve recurring types (§2), and can thus be debugged using a small number of templates (one per fault type). Second, we assume that system logs sufficiently highlight the set of potential parameter values and the relative importance of each; as per §3.1, user reports are often abstract and rarely list parameter values (e.g., switch IDs). Modularization thus shrinks the output space of the first model, simplifying training computationally, regardless of system scale. It also shrinks the input space of the second model, making it less likely to overfit to spurious inputs, which in turn improves accuracy and generalizability.

Challenge 4: Generalizing to new fault locations. Given the scale of production systems, it is infeasible to rely on training data that captures all possible locations of a given fault category. Thus, our model should generalize to *different locations* for fault types seen during training. To aid with such generalization, we convert concrete switch/function ids in the system logs into abstract ids based on the rank order per feature (e.g., queue depth). This allows our models to learn the relevance of a given template or the importance of a particular subsystem based on a stable property like the subsystem's rank on a feature rather than a volatile property, e.g., switch ID (Figure 2).

For example, during template prediction, the model is able to learn about the applicability of a template to the order

Table 2: **Variables in Revelio's ML model. Figure 11 in §A lists example input values for each.**

Name	Description	Example
T	Query template	SELECT *queue_size* FROM *logs* WHERE *switch_id* = _
B	Blanks in T $\{b_1, b_2, ..., b_z\}$	b_i = _ in the above example
U	Query parameters $\{u_1, u_2, ..., u_z\}$	u_i = switch ID
R	User report	"Page is loading slowly"
L	System logs	OpenTracing and Marple logs

statistics (David & Nagaraja, 2004) of feature values across the system, rather than to the numerical or ordinal values of these features at specific subsystems. This is important because, if a given fault occurs at two different locations (both of which warrant the same template), the order statistics of feature values may be correlated, whereas the specific value assignments definitively will not. Similarly, for parameter prediction, ordering information is more robust to the addition, deletion, or restructuring of subsystems.

4 REVELIO'S ML MODEL

To enable Revelio's prediction capabilities, we need to induce a distribution $\mathsf{P}(Q|R; L)$ where Q is a debugging query, R is a user report, and L refers to the system logs (Table 2 lists the model's variables). Once the parameters of this distribution have been learned by maximum likelihood, the distribution allows us to predict the query Q that maximizes $\mathsf{P}(Q|R; L)$. The required training data is a set of triples $\langle R; L; Q\rangle$. While the above formulation seems straightforward, it involves learning a probability distribution over all possible queries and across all tools, which is challenging and needs a substantial amount of data. Thus, we instead split up each query Q into a query template T (e.g., SELECT _ FROM _) and a set of values U (to fill in the blanks). This lets us factorize the prior distribution as:

$$\mathsf{P}(Q|R, L) = \mathsf{P}(T, U|R, L) = \mathsf{P}_1(T|R, L)\mathsf{P}_2(U|T, R, L) \quad (1)$$

To simplify our training further, we make an independence assumption on P_2 by assuming that R is not likely to help predict U (as described in §3.2). Thus, we have:

$$\mathsf{P}_2(U|T, R, L) = \mathsf{P}_2(U|T, L) \quad (2)$$

We make a second independence assumption on the parameters within each blank and further factorize this into a product of distributions over values u_i for each blank b_i in the template T:

$$\mathsf{P}_2(U = \{u_1, u_2, ..., u_z\}|T, L) = \prod_{i \in [1,z]} \mathsf{P}_2(u_i|b_i, T, L) \quad (3)$$

where z is the total number of blanks in the template.

From an inference standpoint, this means we have a 2-phase query generation process: we first generate a query template and then fill in the blanks with appropriate values using the system logs (Figure 3). We next detail how we model each of the distributions (P_1 and P_2), as well as our learning and inference procedures for each.

4.1 Predicting Probabilities for Query Templates (P_1)

Assume the user report R to be in the form of raw text and L to be a vector obtained by concatenating ordered vectors for each feature (Figure 10 in §A) extracted from the system logs (e.g., time-windowed average, minimum queueing delay). Recall from §3.2 that rank ordering per feature in L enables our model to learn about the order statistics of feature values across subsystems, rather than about numerical or ordinal values at specific subsystems (Figure 2). From here, a straightforward way of modeling $\mathsf{P}_1(T|R; L)$ would be to use a multi-label classifier with each template T being a different label. However, as discussed in §3, query templates are structured and made up of smaller atomic components (e.g., IF, MAX statements). In other words, the ASTs of many query templates share common subtrees. Thus, *simply treating each template as an independent output label is wasteful in terms of not sharing statistical strength*.

Therefore, we adopt a different approach to modeling the output templates. In order to preserve the structural aspects in queries, we represent each template T in the form of an abstract syntax tree (AST). Each node in the tree is an operator (e.g., SELECT) and the edges represent how the operators are composed together to form larger trees (Figure 11).

We use a Graph Convolutional Network (GCN) (Kipf & Welling, 2016) to construct a vector representation v_T for each query template's AST. The AST representation of the debugging query is the input to the GCN, with each node mapped to a vector embedding. The GCN, in turn, outputs a vector representation that is directly comparable to the vectors representing system logs. The GCN updates each node's vector representation in the AST by pooling information from all its neighbors and performs this process multiple times, allowing it to combine information from all nodes in the tree. The GCN outputs a vector for each node in the tree – we take the vector of the root node v_T to represent the tree's information.

In parallel, we use a contextual text encoder (BERT) (Devlin et al., 2018) to convert the issue report R into a vector v_R and pass the log L through a linear neural network layer to get a vector v_L. v_R and v_L are concatenated and fed through a non-linear layer followed by a linear layer to get a single vector v_S representing the system state from both internal and external viewpoints. Finally, we use both v_S

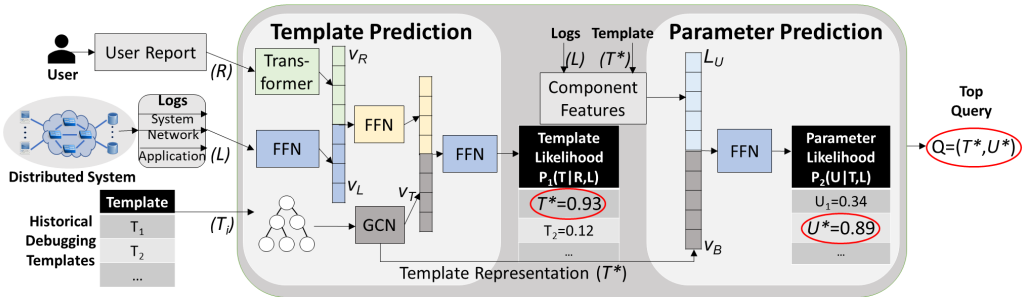

Figure 3: **Overview of Revelio's factorized, 2-phase approach to generating debugging queries for root cause diagnosis.**

and v_T to obtain a measure for the likelihood for the template T being applicable to the debugging scenario $\langle R, L \rangle$ (i.e., the probability of T given R and L). The ideal debugging query paired with the debugging scenario gets a high score (S), while any other query T' paired with the same scenario gets a low score. The sequence of operations is summarized as:

$$
\begin{aligned}
v_T &= \text{GCN}(T)[root] \\
v_R &= \text{BERT}(R) \\
v_L &= \text{LINEAR}(L) \\
v_S &= \text{LINEAR}(\text{RELU}([v_R; v_L])) \\
S(T,R,L) &= \text{LINEAR}(\text{RELU}([v_S; v_T])) \\
\mathsf{P}_1(T|R,L) &= \text{SOFTMAX}(S(T,R,L)) = \frac{e^{S(T,R,L)}}{\sum_{T'} e^{S(T',R,L)}}
\end{aligned}
$$

where $[;]$ represents a concatenation of two or more vectors and $GCN(T)[root]$ represents indexing the output of the GCN to get the vector of the root node.

The above operations represent a continuous flow of information through a *single* DNN whose parameters can be trained, using the training dataset D, through back-propagation and stochastic gradient descent (Goodfellow et al., 2016). We use the following maximization objective to learn the parameters:

$$\max_{\theta} L(\theta) = \sum_{(T,R,L) \in D} \log \mathsf{P}_1(T|R,L) \quad (4)$$

Enumerating all trees T' is intractable, so we employ Noise Contrastive Estimation (NCE) (Gutmann & Hyvärinen, 2010) and draw $m = 2$ negative samples to form each T' to approximate the objective.

4.2 Predicting Values to Fill Query Templates (P_2)

Now that we have a method to pick a template T^*, we must fill in the values for each blank b in T^*. Each template implicitly specifies the type of subsystem that is relevant for the fault at hand. Thus, using the template, we first extract a list of all relevant subsystems from the system logs L. For each subsystem u in this list, we have a feature vector L_u which summarizes all of its logs (Table 4). We also include ranking information $rank_u$ for each feature in L_u (e.g., u's rank in queue depth across all switches). Note that ranks embed the same information as ordering from §4.1. We use these features, along with a vector representation of the blank in the template (described below), to pick the most likely subsystem to fill in the blank.

We feed the template (AST) T^* through the same GCN module as in §4.1 and choose the vector representation for blank b to be the output vector of its corresponding node in the tree. This allows us to represent the requirements of b using the properties of its neighboring nodes in the AST. Our goal is to then pick the *most suitable subsystem* u for the blank, and return the corresponding system identifier (e.g., IP address or port number). We use similar operations to those in §4.1 to pick the most likely u to fill b:

$$
\begin{aligned}
v_b &= \text{GCN}(T)[b] \\
S(u,b,T,L) &= \text{LINEAR}(\text{RELU}(\text{LINEAR}(v_b; L_u; rank_u))) \\
\mathsf{P}_2(u|b,T,L) &= \text{SOFTMAX}(S(u,b,T,L))
\end{aligned}
$$

where $rank_u$ indicates the rank of subsystem u in its subsystem's logs L, based on the feature of interest (e.g., rank of a switch, across all switches, on mean queue depth). We then use an objective similar to Eq. 4 to maximize P_2 over ground truth data and learn the model parameters .

4.3 Inference: choosing the queries

Once each of the two models above have been trained, during inference, we find the combination of query template and query parameters that maximizes the probability that the resulting query would result from the given system state vector. This probability in turn is the product of the two probabilities predicted by each of our models P_1 and P_2.

$$Q^* = (T^*; U^*) = \arg\max_{T;U} \mathsf{P}_1(T|R;L)\,\mathsf{P}_2(U|T;L) \quad (5)$$

We can also pick the top k most relevant queries, rather than just the single most relevant one, using the ranking produced by the probabilities above. If $|T|$ $|U|$ proves to be very large, we can approximate the above by considering only the top few templates according to $\mathsf{P}_1(T|R;L)$.

Implementation details. For all FFN layers in our ranking model, we use two linear layers, each with hidden

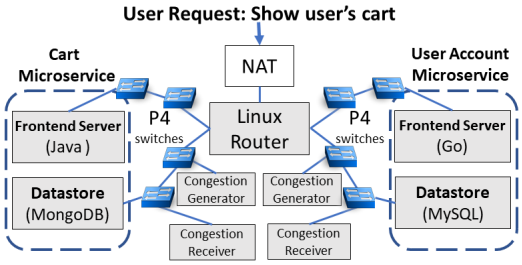

Figure 4: **A slice (2/14 microservices) of our testbed for Sock Shop (Weaveworks, 2017); other apps are shown in §A. Debugging tools and fault injection are omitted for space.**

size 300, along with ReLU non-linearity. The GCN also uses a hidden vector size of 300. We use the Adam optimizer (Kingma & Ba, 2014) with a learning rate of 0.0001.

5 Systems Debugging Testbed

Developing and testing Revelio requires access to a distributed systems environment with debugging data and system logs. Industrial systems (including *Anon*) satisfy these requirements internally (Jackson, 2021; Orate, 2019) but, to our knowledge, no such environment exists for public use. While we were able to analyze debugging reports at *Anon*, we could not access (stored) system logs, precluding our use of Revelio at *Anon*. We instead opt for an extensible in-house testbed (Figure 4) that incorporates state-of-the-art distributed apps, debugging tools, and fault injection.

5.1 Single-Machine Emulation of Distributed Apps

Applications. Our testbed considers three distributed web apps: Reddit (Reddit, 2016) (monolithic), Sock Shop (Weaveworks, 2017) (microservice-based), and Online Boutique (Google, 2019) (microservice-based). For each, we use the publicly available source code that was provided by the corresponding organization and is intended to capture the technologies and architectures employed in production. §A.2.1 provides additional details.

Our goal is to run each application in a distributed and controlled manner, in order to scale to large workloads and deployments, consider broad sets of realistic distributed debugging faults, and ultimately generate complete debugging datasets for Revelio. One approach would be to run each application service on VM instances in the public cloud. However, public cloud offerings typically hide inter-instance network components (e.g., switches) from users, precluding the use of in-network debugging tools (§7).

Instead, we use local emulation whereby we run each subsystem (or service) in a different container on the same machine, and specify the network infrastructure and connectivity between them. We use Containernet (Peuster et al., 2016), an extension of Mininet (Lantz et al., 2010) that can coordinate Docker (Docker.com, 2022) containers, each running on a dedicated core; we assign a separate core for network operation (i.e., P4 (Bosshart et al., 2014) switch simulation). Testbed throughput can be scaled up by using more physical machines through distributed emulation (Wette et al., 2014).

For each application, we configure its subsystem/service containers into a star topology. At the center is a router which can cause layer 3 faults (e.g., firewall configuration errors). Each subsystem is connected to this router via two P4 programmable switches. We set routing rules to ensure that all subsystems are appropriately reachable. Finally, a NAT connects the central router to the host machine's Internet-reachable interface.

5.2 Integrating Debugging Tools

Our testbed has 4 debugging tools (details in §A.2.2):

Marple (Narayana et al., 2017) is a query language for network monitoring that uses SQL-like constructs (e.g., groupby, filter) to support queries that track 1) per-packet and per-switch queuing delays, and 2) user-defined aggregation functions across packets.

tcpdump (Tcpdump.org, 2022) is an end-host network stack inspector which analyzes packets flowing through the host's network interfaces, and supports querying by filtering on packet headers (e.g., by source address).

Uber's Jaeger (Jaegertracing.io, 2022) is a distributed tracing system which follows the OpenTracing specification (Opentracing.io, 2022). Developers embed tracepoints into application source code to log custom state, and then aggregate tracepoint and timing information to understand the flow of user requests across subsystems.

Google's cAdvisor (Google, 2022) profiles the resource utilization of individual containers, logging the following per second: instantaneous CPU usage, memory usage, disk throughput, and total page faults.

5.3 Fault Injection Service

To create data from realistic debugging scenarios, we created an automatic fault injection service. We note that our goal is not necessarily to match the system scale at which production faults were reported, but instead to evoke the user reports, system log patterns, and queries that correspond to the reported fault categories. Our service is guided by our literature survey of production faults and our findings at *Anon* (§2 and §A.1). Specifically, we incorporate faults that cover all of the observed categories, and match the ratios across categories with the data from *Anon* (Table 1). These categories cover both observable performance (i.e., increased response times) and functionality (i.e., miss-

Table 3: **Summary of debugging queries and user reports.**

Metric	Reddit	Sock Shop	Online Boutique
# of Unique Faults	76	102	80
# of Unique Queries	118	320	269
Query Vocabulary Size	60	136	122
Report Vocabulary Size	1040	1327	1258

Table 4: **Metrics in system logs. Marple, Jaeger, and cAdvisor metrics are recorded per-switch, per-function (across Jaeger traces), and per-container respectively; tcpdump is omitted for space.**

Marple	Jaeger	cAdvisor
Packet count	# of accessed variables	CPU utilization
Queue depth	Duration of execution	Memory utilization
N/A	# of exceptions thrown	Disk throughput

ing or inconsistent content, crashes) issues for the apps we consider. Table 10 (in §A) lists the faults we inject.

6 EVALUATION

Data collection. To extract system logs, user reports, and debugging queries from our testbed, we conducted a large-scale data collection experiment on Amazon Mechanical Turk. For each application, we set up an EC2 instance per fault that we consider (Table 3). Each instance runs the entire testbed for that application, with the associated fault injected into it. Application data is collected using scripts included in each application repository.

Our experiment supported only "master" Turk users, and each user was only allowed to participate once per fault+application pair. Each user was assigned to a specific instance/fault at random, and was presented with a UI that pointed to the corresponding instance's frontend web server. Users were asked to perform multiple tasks within each app, including loading the homepage, clicking on item pages or user profiles, adding comments, and adding items to their carts. Prior to the experiment, users were shown example page loads (to ensure familiarity).

For each task, users were asked to report performance and functionality issues via multiple choice and free-form questions. During each experiment, the standard system logs for each testbed tool were collected on the instance (Table 4). We condense and featurize the time-series data for each metric using standard statistics, e.g., min, max, avg. User reports were paired with the associated system logs. We allowed up to 5 concurrent users per instance, and system logs reflect the interactions of all concurrent users. To complete our dataset, for each fault, we generate a debugging query with the appropriate tool that sufficiently highlights the root cause. This query is intended to represent the result of a past, successful, debugging experience. In our experiments, we generated ideal debugging queries using perfect knowledge of the bugs applied to our testbed. Such queries are intended to reflect the outcomes from successful prior debugging; we also generated incorrect queries with negative labels to ensure a balanced dataset. Table 3 summarizes our dataset, and Table 9 (in §A) lists user reports.

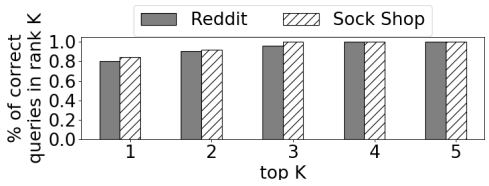

Figure 5: **Cumulative distribution (per app) of the rank of the correct query over our test set of *repeat* faults.**

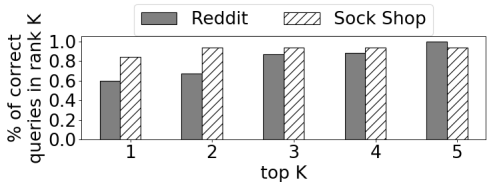

Figure 6: **Cumulative distribution (per app) of the rank of the correct query over our test set of *previously unseen* faults.**

Methodology. We divided the dataset for each application into 53% for training, 13% for validation, and 34% for testing. We further divided our testing data into two test sets, *test_generalize* and *test_repeat*. *test_generalize* evaluates Revelio's ability to generalize to new locations for previously seen fault types, and includes only data for faults that have matching query templates in the training data, but different parameters. *test_repeat* evaluates Revelio's ability to suggest relevant queries for repeat faults, and includes only data for faults that have matching query templates *and* parameters in the training data. All results test the best observed model from the validation set on the test sets.

Metrics. We evaluate Revelio using two main metrics: 1) *rank* of the correct query (i.e., the query that most directly highlights the root cause) among the ordered list of the model's predicted queries, and 2) *top-k accuracy*, defined as the presence of the correct query in the top-k predictions.

Result presentation. Results for Online Boutique are similar to the other two apps, but omitted for ease of presentation. All deep-dive results (§6.2) use Reddit, but the trends hold for all three of the considered apps.

6.1 Evaluating Revelio's Queries

Repeat faults. For each fault in test_repeat, we measured the rank of the correct query in Revelio's predictions. As shown in Figure 5, for 80% of the Reddit test samples, Revelio assigns a rank of 1 to the correct query. Further, for 96% of the Reddit test cases, the correct query is in the top 3 predicted queries. Performance is similar for Sock Shop, with the correct query being in the top 3 100% of the time.

Generalizing to new fault locations. Figure 6 shows re-

Table 5: **Examples where Revelio's top-ranked predictions do not match the ground truth. In all cases, Revelio's top query is highly relevant, but characterizes the fault from an alternate system vantage point. Queries are condensed for ease of disposition.**

Ground Truth Query	Revelio's Top-Ranked Predicted Query

`stream = filter(T, switch==3)`
`result = groupby(stream, [srcip, dstip, srcport, dstport, proto], count);`
*(This **Marple** query highlights the lack of network traffic to/from a (failed) Memcache instance.)*
	`SELECT span FROM spans WHERE name="GET_comments"`
	*(This **Jaeger** query also helps identify the Memcache failure by honing in on the tracepoint with the corresponding 'connection failed' error message.)*

`SELECT * FROM cpu_usage WHERE host="mn.h1"` *(This **cAdvisor** query helps to identify that a given host is consistently running at 100% CPU utilization, and is thus underprovisioned).*	`SELECT span FROM spans WHERE name="_byID"` *(This **Jaeger** query also helps highlight the host's underprovisioned CPU resources by showing the ensuing high function execution times on the host.)*

`SELECT span FROM spans WHERE name="_find_rels"`
*(This **Jaeger** query helps to identify that a bug in a function is resulting in no queries being issued to a MongoDB database.)*
	`stream = filter(T, switch==1)`
	`result = groupby(stream, [srcip, dstip, srcport, dstport, proto], count);`
	*(This **Marple** query shows the lack of network traffic between the host of a buggy function and the database.)*

Table 6: **Impact of different input sources on Revelio's performance. Results list avg rank (% in top-5) and are for Reddit.**

Scenario	test_repeat	test_generalize
User report+system logs	1.33 (100%)	1.97 (100%)
Only system logs	1.86 (100%)	2.29 (90.2%)

sults for the more challenging scenario of new fault locations for repeat fault types (i.e., test_generalize). As shown, Revelio consistently predicts the correct query: Revelio's model assigns a rank of 1 to the correct query 60% and 85% of the time for Reddit and Sock Shop respectively. For both apps, the correct query was *always* in the top-5 predictions.

Benefits of query sequences. We analyzed scenarios in which the correct query was not ranked as 1. We find that in these cases, despite not matching the ground truth, Revelio's highly ranked queries typically relate to the fault at hand, but characterize it from different vantage points; the ground truth query is usually in the top 2-3 queries. Table 5 provides three representative examples. The first example pertains to a fault in which Memcache is down for Reddit. The correct query is a Marple one which tracks packet counts at the switch directly connected to the failed subsystem. However, Revelio's top query used Jaeger to hone in on a tracepoint that contains an error message noting the inability to connect to Memcache. Similarly, in the second example, a Sock Shop subsystem is not provisioned enough CPU resources. The correct query was a cAdvisor one that explicitly tracked the container's CPU usage, but Revelio's top query used Jaeger to track the high residual function execution times for the corresponding microservice.

6.2 Understanding Revelio

Importance of user reports. By default, Revelio's model accepts both natural language user reports and quantitative system logs. To understand the importance of considering user reports in query generation, we evaluated a version of Revelio that excludes user reports from its input set; note that system logs cannot be excluded as they are required for parameter prediction. As shown in Table 6, Revelio significantly benefits from having access to both inputs. For example, on test_generalize, the average rank of the correct query is 1.97 and 2.29 with and without user reports.

Simpler ML approaches. To understand the importance of Revelio's design (§4), we compared it with the following variants: 1) *Revelio_monolithic* uses a single model to output a fully-formed query, 2) *Revelio_no_rank_order* eliminates the rank ordering of features in Revelio's models, and 3) *Revelio_classifier* uses a multi-label classifier to select query templates rather than employing a GCN to construct a vector representation of each template's AST.

Table 7: **Comparison with simpler ML approaches. Results list avg rank (% in top-5) for Reddit.**

Model	test_repeat	test_generalize
Revelio	1.33 (100%)	1.97 (100%)
Revelio_monolithic	17.5 (15.1%)	22.4 (18.5%)
Revelio_no_rank_order	1.29 (100%)	N/A
Revelio_classifier	2.41 (88.7%)	2.69 (86.9%)

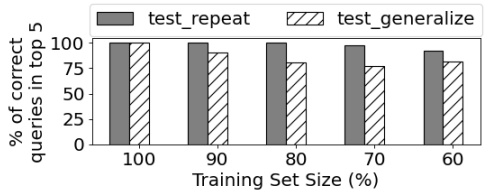

Figure 7: **Top-5 query accuracy when training Revelio on random subsets of the data. Results are for Reddit.**

Table 7 lists our results which highlight three points. First, Revelio outperforms *Revelio_monolithic* on both test sets, highlighting the importance of factorization in terms of simplifying (both computationally and statistically) query prediction, particularly for generalization. Second, by rank ordering feature values, Revelio achieves an average rank of 1.97 for test_generalize; in contrast, *Revelio_no_rank_order* is fundamentally unable to predict templates and parameters (and thus, queries) for repeat fault types in new locations. Third, Revelio's improved performance over *Revelio_classifier* illustrates the importance of extracting semantic information about query structure, which a classifier cannot.

Data and training costs. Training Revelio (using stochastic gradient descent (Ruder, 2016)) took 55 minutes in our experiments, roughly evenly split across the template and parameter prediction models. However, due to computational or security restrictions, organizations may be unable to train on all of the available debugging data, e.g., *Anon* archives years of the debugging data that Revelio requires. Figure 7 shows that Revelio's accuracy remains relatively stable as the training dataset shrinks: with only 60% of

training data, top-5 accuracy drops to 92% and 82% for the test_repeat and (harder) test_generalize test sets.

Additional results. §A.3 presents further results from ablation studies highlighting (1) Revelio's ability to operate across multiple tools simultaneously with low penalty in the efficacy of its outputs (compared to results when using a single model per tool), and (2) the importance of each considered system log feature.

6.3 Developer Study

To evaluate Revelio's ability to accelerate end-to-end root cause diagnosis, we used our testbed to conduct a developer study. 20 developers were presented with the testbed's tools and logs, both with and without Revelio, and were tasked with diagnosing the root cause of multiple high-level user reports. §A.4 describes the study methodology and results. In summary, developers with access to Revelio were able to correctly identify 90% of the root causes (compared to 60% without Revelio), and did so 72% faster.

7 Related Work

We discuss the most closely related approaches here, and present additional related work (e.g., for triaging) in §A.5.

7.1 Debugging Tools for Distributed Systems

There exist dozens of powerful logging and querying tools for distributed systems (Mace et al., 2015; Scott et al., 2016; Sigelman et al., 2010; Mace & Fonseca, 2018; Fonseca et al., 2007; Kaldor et al., 2017), networks (Handigol et al., 2014; Narayana et al., 2016; 2017; Moshref et al.; Tammana et al., 2015; 2016; Tcpdump.org, 2022), and end-host stacks (GNU.org, 2022; Alpern et al., 2000; Netravali & Mickens, 2019; Feldman & Brown, 1988; Ko & Myers, 2008; Lienhard et al., 2008; Viennot et al., 2013; Gyimóthy et al., 1999; Korel & Laski, 1988). However, these tools have two limitations. First, they focus on specific subsystems. Hence, they are not coordinated and lack the full context needed for system-wide debugging. Thus, the cognitive burden of deciding which tools to use, when, and how falls on developers. Revelio interoperates with these tools and alleviates this burden by automatically predicting helpful debugging queries. Second, these tools ignore natural language inputs, despite the fact that they provide debugging insights (Potharaju et al., 2013; Govindan et al., 2016) (§6).

7.2 Leveraging Natural Language Data Sources

Program debugging: NetSieve (Potharaju et al., 2013) uses NLP to parse network tickets by generating a list of keywords and using a domain-specific ontology model to extract ticket summaries from those keywords; summaries highlight potential problems and fixes. While NetSieve automates parsing, much manual effort is still required in (1) offline construction of an ontology model, and (2) determining what constitutes a keyword. In contrast, Revelio's models learn automatically from data, with minimal manual effort, and generate queries for root cause diagnosis rather than potential fixes from a restricted set of actions. Net2Text (Birkner et al.) translates English queries into SQL queries, issues those queries, summarizes the results, and translates them back into natural language for easy interpretation. Revelio, instead, ingests high-level user issues and system logs; the unstructured and abstract nature of this input makes Revelio's problem harder than Net2Text's.

Program analysis and synthesis: NLP techniques have been utilized in multiple aspects of software development (Ernst, 2017). Examples include detecting operations with incompatible variable types (Haq et al., 2015) and converting natural language comments into assertions (Goffi et al., 2016). More recently, NLP has also been used in code generation by converting developer-specified requirements in natural language to structured output in the form of regular expressions (Locascio et al., 2016), Bash programs (Lin et al., 2018), API sequences (Gu et al., 2016), and queries in DSLs (Desai et al., 2016). Though these projects show the potential to extract meaning from natural language debugging data, they are limited to ingesting a single stream of data from a single subsystem. In contrast, Revelio combines and extracts meaning from varied input forms to construct structured queries.

8 Conclusion

Revelio employs ML to generate debugging queries from system logs and user reports to help developers find a problem's root cause faster. Much work remains before this general vision of an ML-enhanced debugging assistant for distributed systems is ready for production use. Notably, Revelio must present a uniform interface to all debugging tools and learn from logs, reports, and queries in an online manner. Despite this, Revelio makes significant progress towards live deployment, demonstrating the importance (in this context) of unified vectors to represent diverse system data, modularity, abstraction to generalize to production systems, and the leveraging query structures in generating debugging queries.

9 Acknowledgements

We thank the MLSys reviewers for their insightful comments, and our the 20 developer study participants for their feedback on Revelio. This work was partially supported by NSF grant CNS-1901510, a Sloan Research Fellowship, and a Cisco Research Award.

REFERENCES

Apache Cassandra. http://cassandra.apache.org/, 2022.

HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/, 2022.

PostgreSQL. https://www.postgresql.org/, 2022.

Pylons. http://www.pylonsproject.org/, 2022.

Messaging that just works – RabbitMQ. https://www.rabbitmq.com/, 2022.

Abuzaid, F., Kraft, P., Suri, S., Gan, E., Xu, E., Shenoy, A., Ananthanarayan, A., Sheu, J., Meijer, E., Wu, X., and et al. DIFF: A Relational Interface for Large-Scale Data Explanation. *Proc. VLDB Endow.*, 12(4): 419–432, December 2018. ISSN 2150-8097. doi: 10.14778/3297753.3297761. URL https://doi.org/10.14778/3297753.3297761.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3 (POPL):1–29, 2019.

Alpern, B., Ngo, T., Choi, J.-D., and Sridharan, M. DejaVu: Deterministic Java Replay Debugger for JalapeÑO Java Virtual Machine. In *Proceedings of OOPSLA*. ACM, 2000.

Appdynamics.com. Application Performance Monitoring & Management — AppDynamics. https://appdynamics.com/, 2022.

Appel, A. W. SSA is Functional Programming. *SIGPLAN Not.*, 33(4):17–20, April 1998.

Arora, N., Bell, J., Ivančiundefined, F., Kaiser, G., and Ray, B. Replay without Recording of Production Bugs for Service Oriented Applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pp. 452–463, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3238186. URL https://doi.org/10.1145/3238147.3238186.

BakIr, G., Hofmann, T., Schölkopf, B., Smola, A. J., and Taskar, B. *Predicting structured data*. MIT press, 2007.

Birkner, R., Drachsler-Cohen, D., Vanbever, L., and Vechev, M. T. net2text: Query-guided summarization of network forwarding behaviors.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

Chen, J., He, X., Lin, Q., Zhang, H., Hao, D., Gao, F., Xu, Z., Dang, Y., and Zhang, D. Continuous incident triage for large-scale online service systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, pp. 364–375. IEEE Press, 2019. ISBN 9781728125084. doi: 10.1109/ASE.2019.00042. URL https://doi.org/10.1109/ASE.2019.00042.

Chen, Y., Griffith, R., Liu, J., Katz, R. H., and Joseph, A. D. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pp. 73–82. ACM, 2009.

Datadoghq.com. Cloud Monitoring as a Service — Datadog. https://www.datadoghq.com/, 2021.

David, H. A. and Nagaraja, H. N. *Order statistics*. John Wiley & Sons, 2004.

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., and Roy, S. Program Synthesis Using Natural Language. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE. ACM, 2016.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2018.

DHH. Postmortem on the read-only outage of Basecamp on November 9th, 2018. https://bit.ly/2S9pq0t, 2018.

Docker.com. Enterprise Container Platform — Docker. https://docker.com/, 2022.

Donohue, B. Instapaper Outage Cause & Recovery. https://medium.com/making-instapaper/instapaper-outage-cause-recovery-3c32a7e9cc5f, 2017.

Dormando. Memcached-a distributed memory object caching system. https://memcached.org/, 2022.

Dropbox. Kelsey Fix shares the story behind Dropbox's largest outage ever. https://bit.ly/2S6p8az, 2019.

Envoyproxy.io. envoy: an open source edge and service proxy, designed for cloud-native applications. https://www.envoyproxy.io/, 2022.

Ernst, M. D. Natural Language is a Programming Language: Applying Natural Language Processing to Software Development. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, pp. 4:1–4:14, 2017.

Feldman, S. I. and Brown, C. B. IGOR: A System for Program Debugging via Reversible Execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD. ACM, 1988.

Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI. USENIX Association, 2007.

GNU.org. GDB: The GNU Project Debugger. `https://www.gnu.org/software/gdb/`, 2022.

Goffi, A., Gorla, A., Ernst, M. D., and Pezzè, M. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016. ACM, 2016.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT press, 2016.

Google. Online Boutique: Sample cloud-native application with 10 microservices showcasing Kubernetes, Istio, gRPC and OpenCensus. `https://github.com/GoogleCloudPlatform/microservices-demo`, 2019.

Google. cAdvisor. `https://github.com/google/cadvisor`, 2022.

Govindan, R., Minei, I., Kallahalla, M., Koley, B., and Vahdat, A. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM. ACM, 2016.

Grafana.com. Grafana Features — Grafana Labs. `https://grafana.com/grafana/`, 2022.

Grpc.io. gRPC: A high performance, open-source universal RPC framework. `https://grpc.io/`, 2022.

Gu, X., Zhang, H., Zhang, D., and Kim, S. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 631–642. ACM, 2016.

Gutmann, M. and Hyvärinen, A. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 297–304, 2010.

Gyimóthy, T., Beszédes, A., and Forgács, I. An Efficient Relevant Slicing Method for Debugging. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, London, UK, UK, 1999. Springer-Verlag.

Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., and McKeown, N. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2014.

Haq, I. U., Caballero, J., and Ernst, M. D. Ayudante: Identifying Undesired Variable Interactions. In *Proceedings of the 13th International Workshop on Dynamic Analysis*, WODA 2015. ACM, 2015.

Icinga.com. Inspect your Entire Infrastructure. `https://icinga.com/`, 2022.

Inc., M. mongoDB: The database for modern applications. `https://www.mongodb.com/`, 2019.

INT. INT-current-spec. `https://p4.org/assets/INT-current-spec.pdf`, 2021.

Jackson, J. Debugging Microservices: Lessons from Google, Facebook, Lyft. `https://bit.ly/2tBS9By`, 2021.

Jaegertracing.io. Jaeger. `https://www.jaegertracing.io/docs/1.26/`, 2022.

Kaldor, J., Mace, J., Bejda, M., Gao, E., Kuropatwa, W., O'Neill, J., Ong, K. W., Schaller, B., Shan, P., Viscomi, B., Venkataraman, V., Veeraraghavan, K., and Song, Y. J. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP. ACM, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Ko, A. J. and Myers, B. A. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE. ACM, 2008.

Korel, B. and Laski, J. Dynamic program slicing. *Information processing letters*, 29(3):155–163, 1988.

Lantz, B., Heller, B., and McKeown, N. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX. ACM, 2010.

Lienhard, A., Gîrba, T., and Nierstrasz, O. Practical Object-Oriented Back-in-Time Debugging. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP, Berlin, Heidelberg, 2008.

Lightstep.com. Simple Observability for Deep Systems. `https://lightstep.com/`, 2022.

Lin, X. V., Wang, C., Zettlemoyer, L., and Ernst, M. D. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation, LREC 2018, Miyazaki, Japan, May 7-12, 2018.*, 2018.

Liu, H., Lu, S., Musuvathi, M., and Nath, S. What Bugs Cause Production Cloud Incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pp. 155–162, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367271. doi: 10.1145/3317550.3321438. URL `https://doi.org/10.1145/3317550.3321438`.

Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N., and Barzilay, R. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, 2016.

Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., and Zhou, Y. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. *SIGOPS Oper. Syst. Rev.*, 42 (2):329–339, March 2008. ISSN 0163-5980. doi: 10.1145/1353535.1346323. URL `https://doi.org/10.1145/1353535.1346323`.

Lyft Engineering. Lyft. `https://eng.lyft.com/tagged/microservices`, 2022.

Mace, J. and Fonseca, R. Universal Context Propagation for Distributed System Instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys. ACM, 2018.

Mace, J., Roelke, R., and Fonseca, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP. ACM, 2015.

McCanne, S. and Jacobson, V. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pp. 2, USA, 1993. USENIX Association.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pp. 3111–3119, 2013.

Moshref, M., Yu, M., Govindan, R., and Vahdat, A. Trumpet: Timely and Precise Triggers in Data Centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM. ACM.

Mysql.com. MySQL. `https://www.mysql.com/`, 2022.

Nagaraj, K., Killian, C., and Neville, J. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pp. 26, USA, 2012. USENIX Association.

Narayana, S., Arashloo, M. T., Rexford, J., and Walker, D. Compiling Path Queries. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.

Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM. ACM, 2017.

Netravali, R. and Mickens, J. Reverb: Speculative debugging for web applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

Newrelic.com. New Relic — Deliver more perfect software. `https://newrelic.com/`, 2022.

Opentracing.io. The OpenTracing project. `https://opentracing.io/`, 2022.

Orate. Ticketmaster Traces 100 Million Transactions per Day with Jaeger. https://bit.ly/39rTn1N, 2019.

Peuster, M., Karl, H., and van Rossem, S. MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pp. 148–153, Nov 2016. doi: 10.1109/NFV-SDN.2016.7919490.

Pham, P., Jain, V., Dauterman, L., Ormont, J., and Jain, N. Deeptriage: Automated transfer assistance for incidents in cloud services. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pp. 3281–3289, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3403380. URL https://doi.org/10.1145/3394486.3403380.

Pingdom.com. Website Performance and Availability Monitoring — Pingdom. https://www.pingdom.com/, 2022.

Potharaju, R., Jain, N., and Nita-Rotaru, C. Juggling the Jigsaw: Towards Automated Problem Inference from Network Trouble Tickets. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating Bugs as Allergies—a Safe Method to Survive Software Failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pp. 235–248, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930795. doi: 10.1145/1095810.1095833. URL https://doi.org/10.1145/1095810.1095833.

Reddit. reddit. https://github.com/reddit/reddit, 2016.

Redis.io. Redis. https://redis.io/, 2022.

Ruder, S. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. URL http://arxiv.org/abs/1609.04747.

Saltside Engineering. Our First Kubernetes Outage — Saltside Engineering. https://engineering.saltside.se/our-first-kubernetes-outage-c6b9249cfd3a, 2017.

Scott, C., Panda, A., Brajkovic, V., Necula, G., Krishnamurthy, A., and Shenker, S. Minimizing Faulty Executions of Distributed Systems. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Technical report, Google, 2010.

Splunk.com. SIEM, AIOps, Application Management, Log Management, Machine Learning, and Compliance — Splunk. https://www.splunk.com/, 2022.

Tammana, P., Agarwal, R., and Lee, M. CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR. ACM, 2015.

Tammana, P., Agarwal, R., and Lee, M. Simplifying Datacenter Network Debugging with Pathdump. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI. USENIX Association, 2016.

Taylor, S. J. and Letham, B. Forecasting at scale. *The American Statistician*, 72(1):37–45, 2018.

Tcpdump.org. TCPDUMP/LIBPCAP public repository. https://www.tcpdump.org/, 2022.

Twitter Engineering. Introducing practical and robust anomaly detection in a time series. https://bit.ly/3oS2Ry9, 2015.

Vagner, K. and Molla, R. After almost 24 hours of technical difficulties, Facebook is back - Vox. https://www.vox.com/2019/3/14/18265793/facebook-app-down-outage-resolved-fixed, 2019.

Viennot, N., Nair, S., and Nieh, J. Transparent Mutable Replay for Multicore Debugging and Patch Validation. In *Proceedings of ASPLOS*, 2013.

Weaveworks. Sock Shop: A Microservices Demo Application. https://microservices-demo.github.io/, 2017.

Wette, P., Dräxler, M., Schwabe, A., Wallaschek, F., Zahraee, M. H., and Karl, H. Maxinet: Distributed emulation of software-defined networks. In *2014 IFIP Networking Conference*, pp. 1–9. IEEE, 2014.

Xu, H., Chen, W., Zhao, N., Li, Z., Bu, J., Li, Z., Liu, Y., Zhao, Y., Pei, D., Feng, Y., et al. Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications. In *Proceedings of the 2018 World Wide Web Conference*, pp. 187–196, 2018.

Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G. R., Zhao, X., Zhang, Y., Jain, P. U., and Stumm, M. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pp. 249–265, USA, 2014. USENIX Association. ISBN 9781931971164.

Zipkin.io. OpenZipkin A distributed tracing system. `https://zipkin.io/`, 2022.

A APPENDIX

A.1 Literature Survey

We surveyed many recent papers and blog posts that document or measure bugs in production settings. Our survey includes major outages in large-scale services (e.g., Dropbox (Dropbox, 2019), Kubernetes (Saltside Engineering, 2017)), bugs in cloud services (e.g., Google (Sigelman et al., 2010), Facebook (Kaldor et al., 2017), Azure (Liu et al., 2019)), and experiences with open source systems (e.g., Cassandra, HDFS (Yuan et al., 2014)). Our survey revealed the following bug categories:

1. **System software and configuration faults.**

   Resource underprovisioning (Kaldor et al., 2017; Saltside Engineering, 2017): In such bugs (e.g., at Facebook (Kaldor et al., 2017)), the containers or VMs running parts of a distributed system are allocated insufficient CPU, memory, disk, or network bandwidth.

   Component failures (Fonseca et al., 2007; Yuan et al., 2014; DHH, 2018; Vagner & Molla, 2019; Dropbox, 2019): Failures are common at scale, and can result from a faulty physical machine, a bug in the machine's hypervisor, or an unduly small amount of memory being allocated to a particular component.

   Subsystem misconfigurations (Yuan et al., 2014; Vagner & Molla, 2019; Liu et al., 2019): Errors in the internal configuration files for a given subsystem are common, especially given complex interoperation with other subsystems. Examples include incorrect hostname mappings that result in improper traffic routing and poorly configured values for timeouts or maximum connection limits (Liu et al., 2019).

2. **Network faults.**

   Network congestion (Kaldor et al., 2017): Within data centers (Kaldor et al., 2017), queues build up at various network locations (e.g., virtual and physical switches) that connect subsystems, either due to temporarily increased application traffic (e.g., TCP incast (Chen et al., 2009)) or cross traffic.

   Incorrect network configuration (Kaldor et al., 2017; Yuan et al., 2014; Vagner & Molla, 2019): Network devices (e.g., firewalls, NATs, switches) between subsystems that communicate via RPCs may be incorrectly configured with forward/drop rules. This could cause unintended forwarding of packets to a destination or incorrect packet dropping.

3. **Application logic faults.**

   Bugs within subsystems (Sigelman et al., 2010; Qin et al., 2005; Lu et al., 2008; Arora et al., 2018; Lu et al., 2005; Donohue, 2017; Saltside Engineering, 2017): Bugs in application logic are prevalent in practice (Netravali & Mickens, 2019; Abuzaid et al., 2018), and can result in a wide range of system effects. For example, certain bugs arise from (accidentally) inverted branch conditions that trigger seemingly inconsistent behavior: an application may traverse an incorrect branch and display incorrect content or result in a program error. In contrast, certain code changes can trigger performance degradations, e.g., if unnecessary RPC calls are generated between microservices.

   Incorrect data exchange formats and values (Liu et al., 2019): Particularly in microservice settings as in Azure services (Liu et al., 2019), bugs can arise if the RPC formats of the sender and receiver do not match. For instance, a change in the API exposed by one microservice could result in a bug if its callers are unaware of this change. Also included in this category are certificate or credential updates that have only been partially distributed (resulting in access control errors).

A.2 Additional Testbed Details

A.2.1 Overview of Applications

Reddit: Reddit is a popular discussion website whose three-tier backend architecture is representative of many distributed applications that utilize the monolithic architectural paradigm. In the front-end tier, HAProxy (hap, 2022) load balances traffic across web servers. The application tier, implemented using the Pylons framework for Python (pyl, 2022), embeds the core application program logic and accesses data objects from the storage tier. The storage tier consists of three data stores: PostgreSQL (pos, 2022) is mainly used as a key/value store for objects such as accounts and comments; Cassandra (cas, 2022) is used as a key/value store for precomputed objects such as comment trees; and Memcache (Dormando, 2022) is used for caching throughout the system. Reddit also uses the RabbitMQ message broker (rab, 2022) to manage asynchronous writes to the storage layer.

Sock Shop: Developed by Weaveworks, Sock Shop is an e-commerce application that employs a microservice-based backend architecture. Sock Shop incorporates 14 different microservices, including a user-facing Node.js frontend microservice, a shopping cart management microservice, a catalog microservice, and so on. Each microservice includes an application server whose logic is implemented in one of a variety of programming languages (e.g., Java, Go), and select microservices additionally operate an individually-managed datastore. For instance, sep-

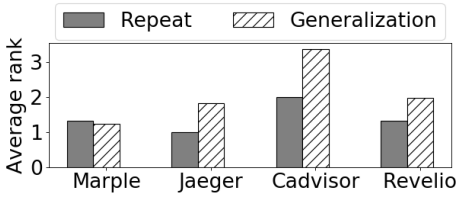

Figure 8: **Comparing the average rank for single-tool and multi-tool versions of Revelio. Results are for Reddit.**

arate MongoDB database instances (Inc., 2019) are used for cart information, processed order transactions, and user profiles, while catalog information is stored in a MySQL database (Mysql.com, 2022). As with Reddit, RabbitMQ manages inter-microservice communication.

Online Boutique: Online Boutique is another microservice-based e-commerce platform from Google that includes 10 distinct microservices that are implemented in Python, Go, C#, Java, and JavaScript. Microservices include a frontend HTTP server (implemented in Go), a payment microservice, a cart microservice, and an ad microservice. Each microservice operates its own datastore, e.g., the cart microservice stores a user's to-be-purchased items in Redis (Redis.io, 2022). Microservices communicate using the gRPC framework (Envoyproxy.io, 2022).

A.2.2 Overview of Debugging Tools

Marple: Marple (Narayana et al., 2017) is a query language for network performance monitoring that uses SQL-like constructs (e.g., groupby, filter). To operate, Marple assigns each network switch and packet a unique ID, and supports queries that track 1) per-packet and per-switch queuing delays, and 2) user-defined aggregation functions across packets. In our implementation, switches log queue depths that each arriving packet encounters, and the packet's 5-tuple (src/dest ip addresses, src/dest ports, and protocol). This is sufficient to track queueing information and high-level statistics such as packet counts. We write queries in Marple to capture and track these values, and then use Marple's compiler to generate P4 programs (Bosshart et al., 2014) that can run directly on our emulated switches. Switches stream query results to a data collection server running on the same host machine for further analysis.

tcpdump: tcpdump is an end-host network stack inspector which analyzes all incoming and outgoing packets across all of the host's network interfaces. tcpdump's command line interface supports querying in the form of packet content filtering (e.g., by hostname, packet type, checksum, etc.), which can be applied at runtime or offline. In our implementation, tcpdump is configured to collect all network packet information in a pcap file, and filters are applied offline.

Jaeger: Uber's Jaeger framework is an end-to-end distributed systems tracing system which, like its predecessors Dapper (Sigelman et al., 2010) and Zipkin (Zipkin.io, 2022), implements distributed tracing according to the OpenTracing specification (Opentracing.io, 2022). With Jaeger, developers embed tracepoints directly into their system source code (or RPC monitoring proxies (Grpc.io, 2022)) and specify custom state (e.g., variable values) to log at each one. By aggregating tracepoint and timing information, Jaeger provides distributed context propagation so developers can understand how data values and control state flows across time and subsystems. We modified each application's source code (application tier for Reddit, and each microservice frontend for Sock Shop and Online Boutique) to include tracepoints for each function accessed during HTTP response generation. As per the examples provided by OpenTracing (Opentracing.io, 2022), at each tracepoint, we log the accessed variables, function execution duration, and any thrown exceptions. During execution, all tracepoint information is sent to a Jaeger aggregation server running on the same host machine for subsequent querying.

cAdvisor: Google's cAdvisor framework profiles the resource utilization of individual containers. To do so, cAdvisor runs in a dedicated container, which coordinates with a Docker daemon running on the same machine to get a listing of all active containers to profile (and the process ids that each owns). With this information, cAdvisor uses the Linux cgroups kernel feature to extract resource utilization information for each container. We use cAdvisor's default configuration, in which the following values are reported every 1 second: instantaneous CPU usage, memory usage, and disk read/write throughput, and cumulative number of page faults. Resource usage information collected by cAdvisor is dynamically sent to a custom logging server running on the same host machine for subsequent querying.

A.3 Additional Results Analyzing Revelio

Multi-tool vs. single-tool models: We performed another ablation study where we compare Revelio when training and testing on logs from each tool together (multi-tool model), and in isolation (single-tool model). For each isolated tool, we prune the training, validation, test_repeat, and test_generalize sets to include only faults pertaining to that tool. Figure 8 shows that the per-tool models achieve better average ranks than the combined (default) model. The reason is that focusing on one tool allows Revelio to predict templates and parameters from a smaller space. However, Revelio pays only a small penalty for operating across debugging tools: the average rank in the combined model is only 33% higher than the best per-tool model. This is key

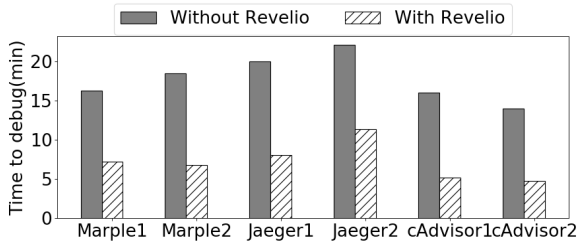

Figure 9: **Summary of time saved in debugging each fault in our developer study. Bars represent average time spent across all developers who correctly identified the root cause.**

Table 8: **Revelio's performance when metrics from system logs are selectively removed. Removed Marple, Jaeger, and cAdvisor features are shown in blue, red, and grey, respectively. Results list avg rank (% in top-5) and are for Reddit.**

Removed Feature	test_repeat	test_generalize
Packet count	2.14 (100%)	7.93 (80.4%)
Queueing delay	2.27 (96.1%)	2.51 (92.4%)
Variable count	1.67 (96.1%)	3.54 (71.7%)
Duration of execution	7.29 (88.2%)	4.11 (89.1%)
CPU utilization	1.65 (96.1%)	4.37 (83.7%)
Memory utilization	1.75 (100%)	2.50 (88.0%)

to Revelio's ability to alleviate the burden of determining which tool to use for a particular scenario.

System log analysis: To understand the relative importance of each metric in the system logs, we evaluated a variety of Revelio models that were trained with each log feature removed, in turn (Table 8). As shown, removing the per-switch *packet counts* from the network logs led to the largest accuracy degradation, with a drop in average rank from 1.97 to 7.93 (for test_generalize). Importantly, removing each considered feature led to marginal degradations in Revelio's performance, highlighting their utility.

A.4 Developer Study

To evaluate Revelio's ability to accelerate end-to-end root cause diagnosis, we used our testbed (§5) to conduct a developer study. Developers were presented with the testbed's tools and logs, both with and without Revelio, and were tasked with diagnosing the root cause of multiple high-level user reports. In summary, developers with access to Revelio were able to correctly identify 90% of the root causes (compared to 60% without Revelio), and did so 72% faster.

Setup and methodology. Our study involved 20 PhD students and postdoctoral researchers in systems and networking. All participants brought their own laptops, but debugging tasks were performed inside a provided VM for uniformity. Prior to the study, the authors delivered a 5-hour tutorial explaining the testbed and Sock Shop UI/code base; the study only involved Sock Shop to ease the developers' ability to become intimately familiar with the application to debug. For each tool (Marple, Jaeger, cAdvisor, tcpdump, Revelio), we described its logs, query language, and interface. Developers were given 1 hour to experiment with the testbed and resolve any questions.

During the study, developers were presented with a series of six debugging scenarios: 2 in-network faults for routing errors and congestion (targeting Marple), 2 system configuration faults for resource underprovisioning and component failures (targeting cAdvisor), and 2 application logic faults for branch condition and RPC errors (targeting Jaeger); we exclude end-host network faults due to time constraints. For each fault type, developers were randomly assigned to debug one fault using only the testbed's tools, and one also using Revelio. Ordering of the faults and tool assignments was randomized across participants to ensure a fair comparison.

For each fault, developers were presented with 1) a user report, 2) system logs for all testbed tools collected during the faulty run, and 3) the faulty testbed code. Developers were given 30 mins to diagnose each fault and provide a short qualitative description of the root cause. For example, a routing configuration error that disconnected Cassandra could be successfully reported as "Cassandra could not receive any network packets, leading to missing page content." When a developer believed she had found the root cause, she informed the paper authors who verified its correctness. If incorrect, the developer was told to keep debugging until a correct diagnosis was generated, or 30 mins elapsed. Developers were unrestricted in their debugging methodologies, e.g., they were not required to use queries, though most did. Without Revelio, developers had to generate any query they wished to issue on their own; with Revelio, developers could generate queries or use the 5 suggested by Revelio.

Revelio's impact on root cause diagnosis. The results of our developer study were promising and suggest that Revelio can be an effective addition to state-of-the-art debugging frameworks in terms of accelerating root cause diagnosis. Across all of the faults, Revelio increased the fraction of developers who could correctly diagnose the faults within the given time frame from 60% to 90%. Further, as shown in Figure 9, Revelio sped up the average root cause diagnosis time by 72% (14 minutes) in cases where the developers were able to report the correct root cause.

After the study, we asked each developer qualitative questions about their experience with Revelio. The most commonly reported benefit of Revelio was in shrinking the set of tools and queries that a developer had to consider. The primary gripe was with respect to Revelio's UI, which is admittedly unpolished. Most importantly, the response to "Would you prefer to use existing systems and networking debugging tools with Revelio?", was "yes" for all 20 participants.

A.5 Additional Related Work

DeepCT (Chen et al., 2019) uses a GRU (Gated Recurrent Unit) and attention-based model to leverage discussions between team members to accurately triage an incident. It incrementally learns knowledge from discussions to better triage the incident. DeepTriage (Pham et al., 2020) leverages both textual and contextual data from incident reports. The textual data contains information like the incident title, summary and the initial discussion entries and the contextual data includes information about the service, severity of the incident, etc. The authors also identify that only few discussion items correspond to triaging and several of the subsequent discussion is on root cause analysis and logging troubleshooting steps which indicates that identifying the root cause requires several manual steps and is time consuming. While triaging identifies the team to detect root cause, it doesn't provide any hints as to the specific metrics and their relation to the subsystems to help developers.

DISTALYZER (Nagaraj et al., 2012) leverages logs printed by distributed systems to identify the strongest association between performance and specific components. These log snippets are returned to the developer to identify the root cause of the incident. Due to the presence of several components in a distributed system and the fact that a bug can manifest itself in logs across different components, it is hard to find the most useful log by association alone and the developer has to still sieve through all the logs.

A.6 Diagrams Illustrating Model Operation/Insights

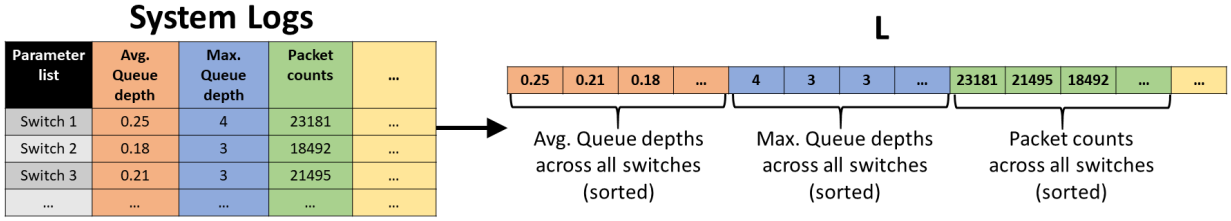

Figure 10: **Example illustrating the generation of system log vector L; for simplicity, the example considers only network logs. Values for each feature (across switches) are first rank ordered, and then the resulting lists are concatenated to form L.**

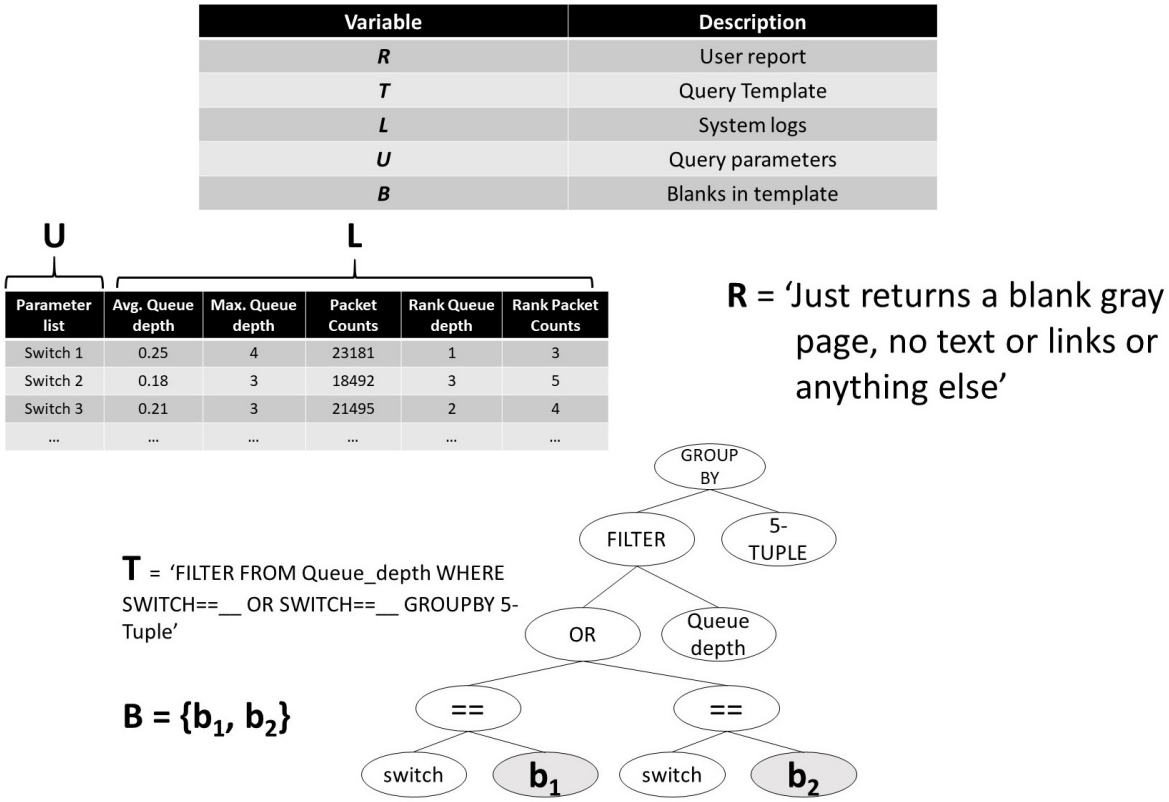

Figure 11: **Example inputs for each input variable in Revelio's model (variables are listed in Table 2 and at the top of this figure). This example is for a network (Marple) query. For the query template (T), the entire tree represents the template, while the parameters to be filled in are shaded in grey.**