CADET: A Systematic Method For Debugging Misconfigurations using Counterfactual Reasoning

Md Shahriar Iqbal∗
University of South Carolina
miqbal@email.sc.edu

Rahul Krishna∗
Columbia University
rahul.krishna@columbia.edu

Mohammad Ali Javidian
Purdue University
mjavidia@purdue.edu

Baishakhi Ray
Columbia University
rayb@cs.columbia.edu

Pooyan Jamshidi
University of South Carolina
pjamshid@cse.sc.edu

Abstract

Modern computing platforms are highly-configurable with hundreds of interacting configuration options. However, configuring these systems is challenging. Erroneous configurations can cause unexpected non-functional faults resulting in significant performance degradation in non-functional system properties like latency, energy consumption, heat dissipation, etc. This paper proposes CADET (short for Causal Debugging Toolkit)—a method that enables users to identify, explain, and fix the root cause of non-functional faults early and in a principled fashion. CADET builds a causal model by observing the performance of the system under different configurations. Then, it uses causal path extraction followed by counterfactual reasoning over the causal model to (a) identify the root causes of non-functional faults, (b) estimate the effects of various configuration options on the non-functional system properties, and (c) prescribe candidate repairs to the relevant configuration options to fix the non-functional faults. We evaluate CADET on 5 highly-configurable software systems deployed on 3 NVIDIA Jetson hardware platforms. We compare CADET with four state-of-the-art machine learning (ML)-based debugging approaches. The experimental results indicate that CADET can find repairs for faults with (on average) 8% better accuracy in multiple non-functional properties 7× faster than the next best performance debugging method.

1 Introduction

Modern computer systems are composed of multiple components, each of which are highly-configurable, and are increasingly being deployed on heterogeneous hardware platforms (e.g., System-on-a-Chip, System-on-Module, IoT devices, cloud platforms) with different deployment configurations (local, distributed, multi-cloud). For example, most modern ML systems, cyber-physical systems, self-driving cars, robotics, and big data systems have such characteristics. The configuration space in such systems is combinatorially large with thousands of software and hardware configuration options that interact non-trivially with one another [1, 2, 3]. Unfortunately, configuring these systems to achieve specific goals is challenging and error-prone. Incorrect configurations (misconfigurations) happen as a result of unexpected interactions between software and hardware configuration options across the system stack resulting in non-functional faults, i.e., faults in terms of non-functional system properties such as latency, energy consumption, and/or heat dissipation. These non-functional faults—unlike regular software bugs—do not cause the system to crash or

∗Joint First author.

exhibit an obvious misbehavior [4, 5, 6]. Instead, misconfigured systems remain operational while being compromised, resulting in severe performance degradation in latency, energy consumption, and/or heat dissipation [7, 8, 9, 10]. The sheer number of modalities of software deployment is so large that exhaustively testing every conceivable software and hardware configuration is impossible.

Consequently, identifying the root cause of non-functional faults is notoriously difficult [11] with as much as 99% of them going unnoticed or unreported for extended durations [12]. Non-functional faults have tremendous monetary repercussions costing companies worldwide an estimated $5 trillion in 2018 and 2019 [13]. They also dominate discussions on online forums where some developers are quite vocal in expressing their dissatisfaction [14, 15]. Therefore, we seek methods that can identify, explain, and fix the root cause of non-functional faults early and in a principled fashion.

**Existing work.** Much recent work has focused on configuration optimization aimed at finding a near-optimal configuration that optimizes a performance objective [16, 17, 18, 19]. Finding the optimum configuration using push-button optimization approaches are not applicable here because we tackle an essentially different problem—to find and repair the root causes of an already observed non-functional fault. The global optima do not give us any information about the underlying interactions between the faulty configuration options that caused the non-functional fault. This information is sought after by developers seeking to address these non-functional faults [4, 20].

Some previous works have used ML-based performance modeling in fixed [21, 22, 23, 24, 25, 26, 27] and variable environments [28, 29, 30, 31]. Several works attempted to debug systems using noisy logs [32], anomaly diagnosis [33, 34], sampling [2], data-driven approaches [35, 36, 37, 38], explanation tables [39], query-based diagnosis [40], statistical debugging and association rule mining based approaches [41, 42, 43, 44, 45, 46], and similarity analysis [47]. These approaches are adept at describing if certain configuration options influence performance, however, they lack the mathematical language to express how or why the configuration options affect performance. Without this knowledge, we risk drawing misleading conclusions. They also require significant time to gather the training samples, and this time grows exponentially with the number of configurations [48, 1]. Recent work has employed causal inference for detecting functional bugs (Holmes [49]) and intermittent failures of databases (AID [50]). These works are orthogonal to performance debugging of highly-configurable systems.

**Limitations of existing work.** In Fig. 1, we present an example to help illustrate the limitations of the current techniques. Here, the observational data gathered so far indicates that a configuration option GPU growth is positively correlated with increased latency (as in Fig. 1a). A black-box ML-model built on this data will, with high confidence, predict that larger GPU growth leads to larger latency. However, this is counter-intuitive because higher GPU growth should, in theory, reduce latency not increase it. When we segregate the same data on swap memory (as in Fig. 1b), we see that there is indeed a general downward trend for latency, i.e., within each group of swap memory, as GPU growth increases the latency decreases. We expect this because GPU growth controls how much memory the GPU can “borrow” from the swap memory. Depending on resource pressure imposed by other host processes, a resource manager may arbitrarily re-allocate some swap memory; this means the GPU borrows proportionately more/less swap memory thereby affecting the latency correspondingly. This is reflected by the data in Fig. 1b. If the ML-based model were to consult the available data (from Fig. 1a) unaware of such underlying causal structure, these models would be incorrect. With thousands of configurations, inferring such nuanced information from optimization or ML-based approaches would require a considerable amount of measurements and extensive domain expertise which can be impractical, if not impossible, to possess in practice.

**Our approach.** In this paper, we propose the use of causal models [51, 52] to express the complex interactions between the configuration options and the performance objectives with causal models—a
We present an instance of a non-functional fault that was reported in the NVIDIA developer forum\(^2\). Here, a developer notices some strange behavior when trying to transplant their code for real-time computation of depth information from stereo-cameras for object detection from NVIDIA Jetson TX1 to TX2. Since TX2 has twice the compute power as TX1, the developer expects to have increased clock speeds and TX2’s use of 128-bit memory bus width versus the 64-bit in TX1\(^6\). RX1 to RX2. Since RX2 has twice the compute power as RX1, the developer expects to have increased clock speeds and RX2’s use of 128-bit memory bus width versus the 64-bit in RX1\(^6\).

2 Motivation — A Real-World Example

We present an instance of a non-functional fault that was reported in the NVIDIA developer forum\(^2\). Here, a developer notices some strange behavior when trying to transplant their code for real-time computation of depth information from stereo-cameras for object detection from NVIDIA Jetson TX1 to TX2. Since TX2 has twice the compute power as TX1, the developer expects to have at least 30% – 40% lower latency in TX2. However, the developer observed that TX2 had at least 30% – 40% lower latency in TX2. However, the developer observed that TX2 had 4× the latency as TX1. To solve this problem, the developer solicits advice from the developer forums. After discussions spanning two days, the developer learns that she/he has made several misconfigurations:

- **Wrong compilation flags**: The compilation does not take into account the microarchitectural differences between the two platforms that may be fixed by setting the -gencode=arch parameter and compiling the code dynamically by disabling the CUDA_USE_STATIC flag. TX1 is based on Maxwell microarchitecture, while TX2 is based on Pascal microarchitecture. These two microarchitectures have significant differences in terms of power usage and compute speed\(^6\).

- **Wrong CPU/GPU clock frequency**: The hardware configuration is set incorrectly. These may be fixed by setting the configuration -nvvpmodel=MAX-N which changes the CPU and GPU clock settings. The MAX-N setting in TX2 provides almost twice the performance of TX1\(^5\) due to increased clock speeds and TX2’s use of 128-bit memory bus width versus the 64-bit in TX1\(^6\).

\(^2\)https://forums.developer.nvidia.com/t/50477

\(^6\)https://github.com/softsys4ai/CADET
CADET: Causal Debugging Toolkit

This section presents a brief description of CADET (outlined in Fig. 2). We gather a few dozen samples of observational data, by measuring the non-functional properties of the system (e.g., latency, etc) under different configuration settings (see ① in Fig. 2) to construct a graphical causal model using the observational data (see ② in Fig. 2). Then, we find paths that lead from configuration options to latency, energy consumption, and thermal output (see ③ in Fig. 2). Next, a query engine generates several counterfactual queries (what-if questions) about specific changes to each configuration option (see ④ in Fig. 2) and finds which of these queries has the highest causal effect on remedying the non-functional fault(s). Finally, we generate and evaluate the new configuration to assert if the newly generated configuration mitigates the non-functional fault(s). If not, we repeat the process by adding this to the current observational data.

Causal structure discovery. In this stage, we express the relationships between configuration options (e.g., CPU freq, etc.) and the non-functional properties (e.g., latency, etc) using a causal model. A causal model is a acyclic directed mixed graph (hereafter, ADMG) [70, 71]. The nodes of the ADMG have the configuration options and the non-functional properties (e.g., latency, etc). Additionally, we enrich the causal graph by including several nodes that represent the status of internal system events, e.g., resource pressure (as in Fig. 1). Unlike configuration options, these system events cannot be modified. However, they can be observed and measured to understand how the causal-effect of changing configurations propagates to latency, energy consumption, or heat dissipation, e.g., resource pressure in Fig. 1 determines how GPU growth affects latency. To build the causal model we gather two dozen samples of observational data (resembling Table 3a). To convert observational data into a causal graph, we use a prominent structure discovery algorithm called Fast Causal Inference (hereafter, FCI) [53]. We picked FCI because it accommodates for the existence of unobserved confounders [53, 72, 73]. This is important because we do not assume absolute knowledge about the configuration space, hence there could be certain configurations we could not modify or system events we have not observed. First, we build a dense graph by connecting all pairs of variables with an undirected edge (as seen in Fig. 3b). Next, we use Fisher’s exact test [74] to evaluate the independence of all pairs of variables conditioned on all remaining variables. Pruning edges between the independent variables results in a skeleton graph as shown in Fig. 3c. Next, we orient undirected edges using edge orientation rules [53, 72, 73, 75] to produce a partial ancestral graph (as in Fig. 3d). We compare all the partially directed edges from the FCI’s PAG (Fig. 3d) with their corresponding counterparts from NOTEARS’ DAG (Fig. 3e). The final causal model would be an ADMG that resembles Fig. 3f.

Causal path extraction. In this stage, we extract paths from the causal graph (referred to as causal paths) and rank them based on their average causal effect on latency, energy consumption, and heat dissipation (our three non-functional properties). A causal path is a directed path originating from either the configuration options or the system events and terminating at a non-functional property.
(i.e., latency, energy consumption, or heat dissipation). To discover causal paths, we backtrack from the nodes corresponding to each non-functional property until we reach a node with no parents. For example, from Fig. 3f, we can extract two paths: (1) GPU growth $\rightarrow$ swap memory $\rightarrow$ Latency, and (2) Resource Pressure $\rightarrow$ swap memory $\rightarrow$ Latency.

A complex causal graph can result in a large number of causal paths. Therefore, we rank the paths in descending order from ones having the highest causal effect to ones having the lowest causal effect on each non-functional property. For further analysis, we use paths with the highest causal effect. To rank the paths, we measure the causal effect of changing the value of one node (say GPU growth or X) on its successor in the path (say swap memory or Z). We express this with the do-calculus notation such as $E[Z | \text{do}(X = x)]$ that represents the expected value of Z (swap memory) if we set the value of the node X (GPU growth) to x. To compute the average causal effect (ACE) of $X \rightarrow Z$, we find the average over all permissible values the node X (GPU growth) can take, i.e., $\text{ACE}(Z, X) = \frac{1}{N} \sum_{x \in X} E[Z | \text{do}(X = x)] - E[Z | \text{do}(X = a)]$.

Here, $N$ represents the total number of values X (GPU growth) can take. If changes in GPU growth result in a large change in swap memory, then the ACE (Z, X) will be larger, indicating that GPU growth on average has a large causal effect on swap memory. The prior equation can be extended to the compute causal effect of a path $P_{ACE}$.

Repairing non-functional faults. In this stage, we use the top K paths with the largest $P_{ACE}$ values to: (a) identify the root cause of non-functional faults; and (b) prescribe ways to fix the non-functional faults. When experiencing non-functional faults, a developer may ask specific queries to CADET and expect an actionable response. For this, we translate the developer’s queries into formal probabilistic expressions that can be answered using the causal paths. We use counterfactual reasoning to generate these probabilistic expressions. To understand query translation, we use the example causal graph of Fig. 3f where a developer observes a latency fault and has the following questions:

Q: “What is the root cause of my latency fault?” To identify the root cause of a non-functional fault we must identify which configuration options have the most causal effect on the performance objective. For this, we use the steps outlined above to extract the paths from the causal graph and rank the paths based on their average causal effect (i.e., $P_{ACE}$ from) on latency. For example, in Fig. 3f we may return the path (say) GPU growth $\rightarrow$ swap memory $\rightarrow$ Latency and the configuration options GPU growth and swap memory both being probable root causes.

Q: “How to improve my latency?” To answer this query, we first find the root cause as described above. Next, we discover what values each of the configuration options must take so that the new latency is better (low latency) than the fault (high latency). For example, we consider the causal path GPU growth $\rightarrow$ swap memory $\rightarrow$ Latency, we identify the permitted values for the configuration options GPU growth and swap memory that can result in a low latency (Y_low) that is better than the fault (Y_high). For this, we formulate a counterfactual expression of the form $\Pr(Y_{\text{low}} | r_{\text{repair}} \neq \text{repair}, Y_{\text{high}} = \text{repair})$ that measures the probability of “fixing” the latency fault with a “repair” (Y_{repairs} = \text{high}) given that with no repair we observed the fault (Y_{repairs} = \text{low}). In our example, the repairs would resemble GPU growth = 0.66 or GPU growth = 0.66, swap memory = 4Gb, etc. We generate a repair set ($\mathcal{R}$) which contains the set of changes where the configurations GPU growth and swap memory are set to all permissible values. Next, we compute the individual treatment effect (ITE) on the latency (Y) for each repair in the repair set $\mathcal{R}$. In our case, for each repair $r \in \mathcal{R}$, ITE is $\text{ITE}(r) = \Pr(Y_{\text{low}} | r_{\text{repair}} \neq \text{repair}, Y_{\text{high}} = \text{repair}) - \Pr(Y_{\text{low}} | r_{\text{repair}} \neq \text{repair}, Y_{\text{low}} = \text{repair})$. ITE measures the difference between the probability that the latency is low after a repair r and the probability that the latency is still high after a repair r. To find the most useful repair ($\mathcal{R}_{\text{best}}$), we find the repair with the largest (positive) ITE, i.e., $\mathcal{R}_{\text{best}} = \arg\max_{r \in \mathcal{R}} \text{ITE}(r)$. This provides the developer with a possible repair for the configuration options that can fix the latency fault.

Incremental learning. In this stage, we generate a new configuration using the recommended repairs from the $\mathcal{R}_{\text{best}}$ value. We reconfigured the system with the new configuration and we observe the system behavior. If the new configuration resolves the non-functional fault, we return the recommended repairs to the developer. Since the causal model uses limited observational data, there may be a discrepancy between the actual performance of the system after the repair and the value of the estimation from $\mathcal{R}_{\text{best}}$ derived from the current version of the causal graph. The more accurate the causal graph, the more accurate the recommended configuration will be [53, 72, 73, 75, 76]. Therefore, in case our repairs do not fix the faults, we update the observational data with this new
configuration and repeat the process. Over time, the estimations of the causal effects will become more accurate. We terminate the incremental learning once we achieve the desired performance.

4 Case Study: Latency Fault in TX2

This section revisits the real-world latency fault previously discussed in §2. For this study, we reproduce the developers’ setup to assess how effectively CADET can diagnose the root-cause of the misconfigurations and fix them. For comparison, we use BugDoc (an ML-based diagnosis tool) and the recommendations by the domain experts on the forum. Fig. 4 illustrates our findings. We find that CADET could diagnose the root cause of the misconfiguration and recommends a fix within 24 minutes. Using the recommended configuration fixes from CADET, we achieved a frame rate of 26 FPS (53% better than TX1 and 6.5× better than the fault). This exceeds the developers’ initial expectation of 30 – 40% (or 22 – 24 FPS). BugDoc performed worse than CADET (21% improvement over TX1) while taking 3.5 hours (time mostly spent on collecting training samples to train internal decision tree) and changed several unrelated configurations (depicted by ✗) not endorsed by the domain experts.

Why CADET works better (and faster)?

CADET discovers the misconfigurations by constructing a causal model (a simplified version of this is shown in Fig. 4). This causal model rules out irrelevant configuration options and focuses on the configurations that have the highest (direct or indirect) causal effect on latency, e.g., we found the root cause CUDA STATIC in the causal graph which indirectly affects latency via context-switches (an intermediate system event); this is similar to other relevant configurations that indirectly affected latency (via energy consumption). Using counterfactual queries, CADET can reason about changes to configurations with the highest average causal effect (last column in Fig. 4). The counterfactual reasoning occurs with no additional measurements, significantly speeding up inference.

Together, the causal model and the counterfactual reasoning enable CADET to pinpoint the configuration options that were misconfigured and recommend a fix for them promptly. As shown in Fig. 4, CADET accurately finds all the configuration options recommended by the forum (depicted by ✗ in Fig. 4). Further, CADET recommends fixes to these options that result in 14% better latency than the recommendation by domain experts in the forum. More importantly, CADET takes only 24 minutes (vs. 2 days of forum discussion) without modifying unrelated configurations.

5 Evaluation

Experimental Setup. This study uses three NVIDIA Jetson platforms: TX1, TX2, and XAVIER and five software systems on each platform: (1) An image recognition system with Xception to classify 5000 images from the CIFAR10 dataset [78]; (2) an NLP system with BERT to perform sentiment analysis on 10000 reviews from the IMDb dataset [79]; (3) An RNN based voice recognition system with DeepSpeech on 5 seconds long audio files; (4) SQLite, a database management system, to perform read, write, and insert operations; and (5) x264 video encoder to encode a video file of size 11MB with a resolution of 1920 x 1080. We use 28 configuration options that include 10 software options, 8 OS/Kernel options, and 10 hardware options. We curate a non-functional faults dataset, called the JETSON_FAULTS dataset, and ground truth for each observed non-functional faults for each of the software and hardware system used in our study. We create a ground-truth data by measuring configurations for a fixed budget of 24 hours and identifying their root-causes manually for each fault by selecting
the configuration with the highest performance gain from the fault. By definition, non-functional faults have latency, energy consumption, and heat dissipation that take tail values [11, 80], i.e., they are worse than the 99th percentile. We filter our data set to find the configurations that result in tail values for latency, energy consumption, and/or heat dissipation and label these configurations as ‘faulty’. We evaluate the predicted root-causes in terms of accuracy (Jaccard similarity). To compute accuracy, we compare the set of configuration options identified by CADET to be the root cause with the true root-cause obtained from the ground truth data. To assess the quality of fixes, we measure the percentage improvement (gain %) after applying the recommended repairs using $\Delta_{gain}$. We prefer higher accuracy and gain.

Results. We compare CADET with four state-of-the-art ML-based methods for fault diagnostics, namely: DELTADEBUGGING [63], CBI [41], BUGDOC [42], and ENCORE [43]. For all methods, we set a maximum budget of 4 hours. All methods require some initial observational data to operate. Within the budget, CADET samples 25 initial observational data to incrementally generate, evaluate, and update the causal model with candidate repairs. Other methods require a large and diverse pool of observational data to train. However, collecting observational data is expensive and time-consuming. Therefore, we use the entire budget of 4 hours to generate random configuration samples to train ML-based methods. We assess the effectiveness of diagnostics for ‘single-objective’ non-functional faults, i.e., faults that occur only in one of latency, energy consumption, or heat dissipation. For brevity, we evaluate latency faults in TX2, energy consumption faults in XAVIER, and heat dissipation faults in TX1. Our findings generalize over other hardware. Table 1 summarizes the effectiveness of CADET over other ML-based fault diagnosis approaches. We observe the following:

- **Accuracy and gain.** CADET significantly outperforms ML-based methods in all cases. For example, in SQLite database management system on TX2, CADET achieves 14% more accuracy compared to BUGDOC (best among the remaining ML-based approaches). We observe similar trends in energy faults, i.e., CADET outperforms other methods in all cases. CADET can recommend repairs for faults that significantly improves latency and energy usage. Applying the changes to the configurations recommended by CADET increases the performance drastically. We observed latency gains as high as 81% (22% more than BUGDOC) on TX2 and energy gain of 83% (32% more than BUGDOC) on XAVIER for image recognition.

- **Wallclock time.** CADET can resolve misconfiguration faults significantly faster than ML-based approaches. In Table 1, the last two columns indicate the time taken (in hours) by each approach to diagnosing the root cause. We find that while other approaches use the entire budget of 4 hours to diagnose and resolve the faults, CADET can do so significantly faster before the maximum budget is exhausted, e.g., CADET is 40× faster in diagnosing and resolving faults in energy usage for x264 deployed on XAVIER and 20× faster in diagnosing latency faults for NLP task on TX2. ML-based methods require a large number of initial observational data for training. They spend most of their allocated 4-hour budget on gathering these training samples. In contrast, CADET starts with only 25 samples and uses incremental learning to judiciously update the casual graph with new configurations until a repair has been found.

Discussion. Table 1 shows that image recognition, NLP and speech recognition deep neural network (DNN) systems had the most improvements with CADET compared to x264 and SQLite. Misconfigurations affecting the onboard GPU lead to severe degradation in latency and energy usage. Since DNN relies on GPU to optimize the operations, it must be configured appropriately to leverage the full hardware potential. Other applications were less sensitive to such misconfigurations. Further, all methods found it difficult to discover and resolve thermal faults. While CADET outperformed other methods, the overall accuracy, and gain were lower than those for latency and energy consumption faults. We believe there are two reasons for this: (1) The workloads exercised in

| Table 1: Efficiency of CADET compared to other approaches. Cells highlighted in blue indicate maximum improvement over faults. CADET achieves better performance overall and is much faster. |

<table>
<thead>
<tr>
<th>Latency</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX</td>
<td>NLP</td>
</tr>
<tr>
<td>CADET</td>
<td>84</td>
</tr>
<tr>
<td>WALL</td>
<td>82</td>
</tr>
<tr>
<td>AVIER</td>
<td>81</td>
</tr>
<tr>
<td>CBI</td>
<td>80</td>
</tr>
<tr>
<td>UG</td>
<td>79</td>
</tr>
<tr>
<td>ORE</td>
<td>78</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TX</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLP</td>
<td>81</td>
</tr>
<tr>
<td>Speech</td>
<td>80</td>
</tr>
<tr>
<td>x264</td>
<td>79</td>
</tr>
<tr>
<td>SQLite</td>
<td>78</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TX</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLP</td>
<td>81</td>
</tr>
<tr>
<td>Speech</td>
<td>80</td>
</tr>
<tr>
<td>x264</td>
<td>79</td>
</tr>
<tr>
<td>SQLite</td>
<td>78</td>
</tr>
</tbody>
</table>

Discussion. Table 1 shows that image recognition, NLP and speech recognition deep neural network (DNN) systems had the most improvements with CADET compared to x264 and SQLite. Misconfigurations affecting the onboard GPU lead to severe degradation in latency and energy usage. Since DNN relies on GPU to optimize the operations, it must be configured appropriately to leverage the full hardware potential. Other applications were less sensitive to such misconfigurations. Further, all methods found it difficult to discover and resolve thermal faults. While CADET outperformed other methods, the overall accuracy, and gain were lower than those for latency and energy consumption faults. We believe there are two reasons for this: (1) The workloads exercised in
his work did not significantly heat the system; and (2) the thermal measurements were taken in a controlled environment (indoor in a stable temperature), as a result, the variance temperature was relatively lower.

6 Conclusion
Modern computer systems are highly-configurable with thousands of interacting configuration options with complex performance behavior. Misconfigurations in these systems can elicit complex interactions between software and hardware configuration options resulting in non-functional faults. We propose CADET (short for Causal Debugging Toolkit), a novel approach for diagnostics that learns and exploits the causal structure of configuration options, system events, and performance metrics. Our evaluation shows that CADET effectively and quickly diagnoses the root cause of non-functional faults and recommends high-quality repairs to mitigate these faults.

Acknowledgments. We like to acknowledge Christian Kästner, Sven Apel, Tianyin Xu, Vivek Nair, Jianhai Su, Miguel Velez, Mohsen Amini Salehi and Tobias Dürschmid for their valuable feedback and suggestions in improving the paper. This work was partially supported by NASA (RASPERRY-SI Grant Number 80NSSC20K1720) and NSF (SmartSight Award 2007202).

References


Fei Wu, Pranay Anchuri, and Zhenhui Li. Structural event detection from log messages. In 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1175–1184, 2017.


