

CADET: A Systematic Method For Debugging Misconfigurations using Counterfactual Reasoning

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

Baishakhi Ray
 Columbia University
 rayb@cs.columbia.edu

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

Mohammad Ali Javidian
 Purdue University
 mjavidia@purdue.edu

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

Abstract

Modern computing platforms are highly-configurable with thousands of interacting configurations. However, configuring these systems is challenging. Erroneous configurations can cause unexpected non-functional faults. This paper proposes CADET (short for Causal Debugging Toolkit) 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 configurable parameters on the performance objective(s), and (c) prescribe candidate repairs to the relevant configuration options to fix the non-functional fault. We evaluated CADET on 5 highly-configurable systems deployed on 3 NVIDIA Jetson systems-on-chip. We compare CADET with state-of-the-art configuration optimization and ML-based debugging approaches. The experimental results indicate that CADET can find effective repairs for faults in multiple non-functional properties with (at most) 17% more accuracy, 28% higher gain, and 40× speed-up than other ML-based performance debugging methods. Compared to multi-objective optimization approaches, CADET can find fixes (at most) 9× faster with comparable or better performance gain. Our case study of non-functional faults reported in NVIDIA’s forum show that CADET can find 14% better repairs than the experts’ advice in less than 30 minutes.

1 Introduction

Modern computing systems are highly configurable and can seamlessly be deployed on various hardware platforms, eg., SoCs, cloud systems, etc [41]; and under different environmental settings, eg., UAV, self-driving cars, robotic systems, etc [58, 73]. The configuration space is combinatorially large with 100s if not 1000s of software and hardware

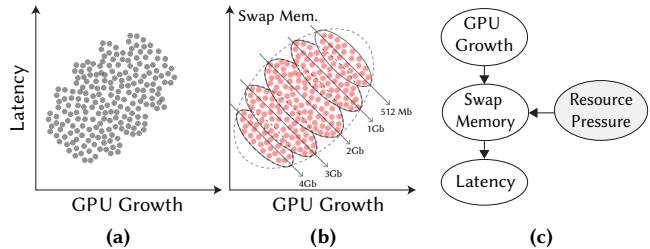


Figure 1. Observational data (in Fig. 1a) (incorrectly) shows that high GPU growth leads to high latency. The trend is reversed when the data is segregated by swap memory (Fig. 1b).

configuration options that interact non-trivially with one another [40, 62, 95]. Unfortunately, configuring these systems to achieve specific goals is challenging and error prone.

Incorrect configuration (*misconfiguration*) elicits unexpected interactions between software and hardware resulting *non-functional faults*, i.e., faults in *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 exhibit an obvious misbehavior [70, 78, 88]. Instead, misconfigured systems remain operational while being compromised, resulting severe performance degradation in latency, energy consumption, and/or heat dissipation [16, 66, 69, 80]. 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 [37] with as much as 99% of them going unnoticed or unreported for extended durations [6].

Non-functional faults have tremendous monetary repercussions costing companies worldwide an estimated \$5 trillion USD in 2018 and 2019 [36]. They also dominate discussions on online forums where some developers are quite vocal in expressing their dissatisfaction. For example, one developer on NVIDIA’s developer forum bemoans: “I am quite upset with CPU usage on TX2 [10],” while another complained, “I don’t think it [the performance] is normal and it gets more and more frustrating [9].” Crucially, these exchanges provoke other unanswered questions, such as, “what would be the

*Joint First Author

effect of changing another configuration ‘X’? [4]. Therefore, we seek methods that can *identify, explain, and fix* the root cause of non-functional faults early in a principled fashion. **Existing work.** Much recent work has focused on configuration optimization, which are approaches aimed at finding a configuration that optimizes a performance objective [32, 49, 50, 68, 82, 93, 96, 101]. 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 does 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 [78, 87].

Some previous work have used machine learning based performance modeling approaches [38, 82, 83, 89]. 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 [53, 95].

Limitations of existing work. In Fig. 1, we present an example to help illustrate the limitations with current techniques. Here, the observational data gathered so far indicates that GPU growth is positively correlated with increased latency (as in Fig. 1a). An ML-model build 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 trend downward 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 factors, these models would be incorrect. With thousands of configurations, inferring such nuanced information from optimization or ML based approaches would require 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 [74, 76] to express the complex interactions between the configurations and the performance objectives with a rich graphical representation. Then, using counterfactual

reasoning [75] we identify the root cause of non-functional faults and prescribe repairs to fix the misconfigured options.

To this end, we design, implement, and evaluate CADET (short for Causal Debugging Toolkit). CADET uses causal structural discovery algorithms [86, 102] to construct a causal model using observational data. Then, it uses counterfactual reasoning over the causal model to: (a) identify the root causes of non-functional faults, (b) estimate the effects of various configurable parameters on the non-functional properties(s), and (c) prescribe candidate repairs to the relevant configuration options to fix the non-functional fault. In the example of Fig. 1, CADET constructs a *causal model* from observational data (as in Fig. 1c). This causal model indicates that GPU growth indirectly influences latency (via a swap memory) and that the configuration options may be affected by certain other factors, e.g., resource manager allocating resources for other process running on the host. CADET uses counterfactual questions such as, “what is the effect of GPU growth on latency if the available swap memory was 2Gb?” to diagnose the faults and recommend changes to the configuration options to mitigate these faults.

Evaluation. We evaluate CADET on 5 real world highly configurable systems deployed on 3 architecturally different NVIDIA Jetson systems-on-chip. We compare CADET with state-of-the-art configuration optimization and ML-based performance debugging approaches. Overall, we find that CADET is (at most) 40× faster with 13% better accuracy and 32% higher gain than the next best ML-based approaches in some cases. Compared to single-objective optimization approaches, CADET can find repairs to misconfigurations 8× faster while performing as well as (or better than) the best configuration discovered by the optimization techniques. Further, CADET can find effective repairs for faults in multiple performance objectives (i.e., latency, energy consumption, and heat dissipation) with accuracy and gain as high as 33% and 32%, respectively, than other ML-based performance debugging methods. Compared to multi-objective optimization approaches, CADET can find repairs 9× faster while having similar performance gain. Finally, with a case study, we demonstrate that CADET finds 14% better repairs than the experts’ advice in 24 minutes.

2 Motivation

2.1 A real-world example

We present an instance of a non-functional fault that was reported in the NVIDIA developer forum¹. Here, a developer notices some strange behavior when trying to transplant their DNN based object detection framework 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, they observed that TX2 had 4× the latency as TX1. The developer uses a reliable

¹<https://forums.developer.nvidia.com/t/50477>

reference implementation of their code and is puzzled by the occurrence of this fault.

To solve this problem, the developer solicits advice from the developer forums. After discussions spanning two days, they learn that they have made several misconfigurations:

- *Wrong compilation flags*: Their compilation does not take into account the microarchitectural differences between the two platforms. These may be fixed by setting the correct microarchitecture with `-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 heat dissipation, power usage, and compute speed [33]. A software built on the new platform that fails to account for these differences would lead to high latency.
- *Wrong CPU/GPU clock frequency*: Their hardware configuration is set incorrectly. These may be fixed by setting the configuration `-nvmodel=MAX-N` which changes the CPU and GPU clock settings. The Max-N setting in TX2 provides almost twice the performance of TX1 [2] due to a number of factors including increased clock speeds and TX2's use of 128-bit memory bus width versus the 64-bit in TX1 [33].
- *Wrong Fan modes (hardware layer)*: Their fan modes need to be configured correctly to account for higher CPU/GPU clock speeds. If the fan modes are not configured to be sufficiently high, TX2 will thermal throttle the CPU and GPU to prevent overheating [5] and invariably increasing the latency [17].

Above example typifies the general nature of discussions on developer forums. Below, we study several other examples of non-functional faults from NVIDIA forums with the goal of understanding and summarizing the key challenges.

2.2 Manual Study

We study NVIDIA's Jetson System-on-Module (SOM) — a widely used high-performance low-power computing platform for intensive AI applications. Each Jetson device is complete with CPU, GPU, PMIC, DRAM, and flash storage. They are highly extensible allowing users to compose a custom system around them to meet specific needs. Their extensibility exposes them to frequent misconfigurations resulting in non-functional faults. In this sense, the Jetson SOMs are a microcosm of broader highly configurable systems.

NVIDIA's developer forum is a preferred destination for users to discuss frequent problems with Jetson devices and potential solutions for them. These discussions, moderated by domain experts, offer a glimpse into the challenges faced by the developers and how they are remedied.

We extract all posts pertaining to NVIDIA Jetson Platforms after 2017 with a focus on posts detailing performance issues. For this, we filter the posts containing at least one of the following keywords: 'performance', 'hardware', 'hw', 'issue', 'problem', and 'fault'. We randomly sample 1000 posts and

inspect them manually². We exclude posts with no active developer participation. We also ensure that (a) the issue in the post is partially or fully resolved; (b) the authors of the issue mention the performance impact and clarify their expectations; (c) the issue is indeed related to a non-functional fault, i.e., it is not due to external factors such as component failure, firmware issue, etc. Finally, we categorize each post based on the specific hardware. In total, we identified 47 performance issues with the following general takeaways:

Takeaway 1. *There are different types of non-functional faults prevalent across all hardware.* There were three types of non-functional faults: (i) latency (20/47, 42%), (ii) energy consumption (17/47, 36%), and (iii) thermal faults (9/47, 22%). The total number of non-functional faults remained roughly the same across hardware. However, certain non-functional faults were more prevalent in one hardware over another. TX1 (more compact SoC) was susceptible to thermal issues, TX2 was susceptible to high latency, and XAVIER (with more powerful CPU/GPU) was most susceptible to energy consumption issues due to suboptimal energy management [27].

Takeaway 2. *Non-functional faults interact with one another.* In 12 out of 47 cases (i.e., 25%), developers experienced multiple types of non-functional faults simultaneously. This indicates that non-functional faults are correlated with one another. Intuitively, increased energy consumption would correspond to a proportional increase in heat radiated. Likewise, reducing the latency would require increasing core frequency which would result in more energy consumed.

Takeaway 3. *Non-functional faults take a long time to resolve.* 29 out of 47 cases (i.e., 61%) took more than 1 week to resolve. The average time to resolve an issue was 5 weeks with some issues taking up to 11 months to resolve [3, 7].

3 Causal Inference

Reusing the example from Fig. 1, we provide a brief summary of causal inference. Let us assume we have gathered several samples of GPU growth, swap memory, and latency. If we are interested in how latency behaves given GPU growth, then, this can be formulated in three ways: observational, interventional, and counterfactual [74, 77].

3.1 Observation and Intervention

In the *observational* formulation, we measure the distribution of a target variable (e.g., latency Y) given that we *observe* another variable GPU growth (X) takes a certain value 'x' (i.e., $X = x$), denoted by $Pr(Y | X = x)$. This is a familiar expression that, given adequate data, state-of-the-art supervised ML algorithms can fully estimate.

The *interventional* inference tackles a harder task of estimating the average effects of deliberate actions. For example, in Fig. 1, we measure how the distribution of Latency (Y) would change if we (artificially) intervened during the data gathering process by forcing the variable GPU growth (X)

²Filtered posts: <https://figshare.com/s/8644807100a9ba812712>

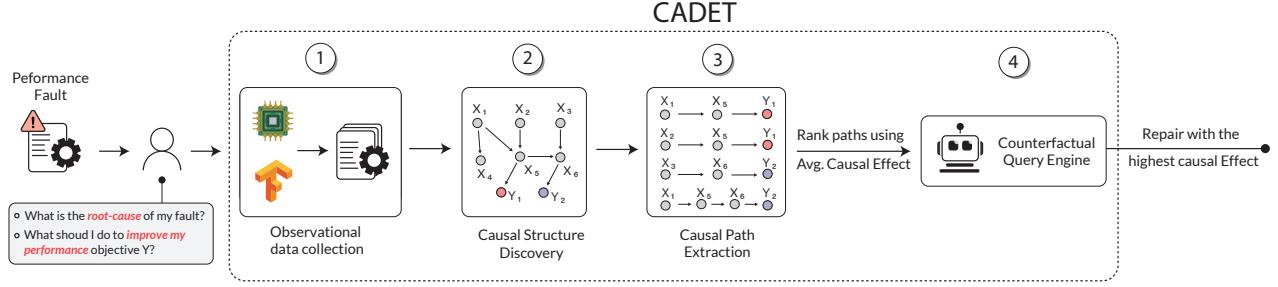


Figure 2. Overview of CADET.

to take a certain value ‘ x ’, i.e., $X = x$, but otherwise retain the other variables (i.e., swap memory) as is. Note that we merely simulate the act of setting the variable GPU growth (X) to take a certain value instead of gathering additional data after making this change. To perform such reasoning, we use causal models and *do-causal calculus* [74].

3.2 Counterfactual Inference

These problems involve probabilistic answers to “what if?” questions. They pertain to alternate possibilities that are counter to the current facts (hence counterfactual) and their consequences. For example, in Fig. 1, a counterfactual question may ask:

Given that we observed a high latency, and given that GPU growth was set to (say) 33%, and given everything else we know about the circumstances of the observation (i.e., available swap memory), what is the probability of decreasing latency, had we set the GPU growth to 66%?

In other words, we are interested in a scenario where:

- We *hypothetically* have low latency;
- Conditioned on the following events:
- We *actually observed* high latency;
- GPU growth was initially set to 33%;
- We *hypothetically* set the new GPU growth to 66%; and
- Other circumstances (available swap memory, resource load, etc) remain the same;

Formally we represent this, by abusing notations, as:

$$\Pr(\hat{Y} = y_{low} \mid \hat{X} = 0.66, X = 0.33, Y = y_{high}, Z) \quad (1)$$

Here, Y stands for observed latency, X stands for current GPU growth, and Z stands for current swap memory. The variables \hat{X} and \hat{Y} are as yet unobserved (or hypothetical), i.e., these are variables that are either being predicted for ($\hat{Y} = y_{low}$) or being *hypothetically* changed to a new value ($\hat{X} = 0.66$).

Questions of this nature require precise mathematical language lest they will be misleading. For example, in Eq. (1), we are simultaneously conditioning on two values of GPU growth (i.e., $\hat{X} = 0.66$ and $X = 0.33$). Traditional machine learning approaches cannot handle such expressions. Instead, we must resort to causal models to compute them.

Computing counterfactual questions using causal models involves three steps [74, Section 7]: (i) *Abduction*: where we

update our model of the world using observational data; (ii) *Action*: where we modify the model to reflect the counterfactual assumption being made; and (iii) *Prediction*: where we use the obtained modified model from the previous step to compute the value of the consequence of the counterfactual.

4 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) of the system under different configuration settings (see ① in Fig. 2) to construct a graphical causal model using the observational data (see ② in Fig. 2 and §4.1). Then, we find paths that lead from configuration options to latency, energy consumption, and thermal output (see ③ in Fig. 2 and §4.2). Next, a query engine generates a number of counterfactual queries, (what-if questions) about specific changes to each configuration option (see ④ in Fig. 2 and §4.3) 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 (see §4.4).

4.1 Causal Graph 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), i.e., an acyclic graph consisting of directed and bidirected edges representing the direction of causal relationships [29, 79]. 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 systems 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.

Data collection. To build the causal model we gather two dozen samples of observational data (resembling Table 3a). It consists of configuration options set to randomly chosen values; various system events occurring when these configurations were used; and the measure of latency, energy consumption, and heat dissipation of the configurations.

Fast Causal Inference. To convert observational data in to a causal graph, we use a prominent structure discovery algorithm called Fast Causal Inference (hereafter, FCI)[86]. We picked FCI because it accommodates for the existence of unobserved confounders [35, 72, 86], i.e., it operates even when there are unknown variables that have not been, or cannot be, measured. This is important because we do not assume absolute knowledge about configuration space, hence there could be certain configurations we could not modified, or system events we have not observed.

FCI operates in three stages. First, we construct a fully connected undirected graph where each variable is connected to every other variable. Second, we prune away spurious edges to produce a skeleton graph. Finally, we orient undirected edges using prescribed edge orientation rules³ to produce a *partial ancestral graph* (or PAG). A PAG contains 4 types of (partially) directed edges:

- $V_1 \rightarrow V_2$ indicates that vertex V_1 causes V_2 .
- $V_1 \leftarrow V_2$ indicates that vertices V_1 and V_2 are causally linked but the directionality is determined by possible *unmeasured confounder* between the vertices.
- $V_1 \circ \rightarrow V_2$ indicates that V_1 causes V_2 , but there could be an *unmeasured confounder* that affects V_1 .
- $V_1 \circ \circ \rightarrow V_2$ indicates that vertices V_1 and V_2 are correlated but the data is insufficient to detect causality.

Example. Fig. 3 shows a simple example with four variables: GPU growth, swap memory, resource pressure, and latency. 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 [21] to evaluate the independence of all pairs of variables conditioned on all remaining variables. In our example, we identify the following conditionally independent variables:

- Latency $\perp\!\!\!\perp$ Resource pressure | swap memory;
- GPU growth $\perp\!\!\!\perp$ Resource pressure | swap memory; and
- GPU growth $\perp\!\!\!\perp$ Latency | swap memory and also Resource pressure;

Pruning edges between the independent variables results in skeleton graph as shown in Fig. 3c. Next, we orient undirected edges using edge orientation rules [20, 35, 72, 86] to produce a partial ancestral graph (as in Fig. 3d).

Resolving partially directed edges. The PAG obtained from FCI may have partially directed edges (i.e., $\circ \rightarrow$ and

$\circ \circ \rightarrow$). These must be fully resolved (directed) in order to generate an ADMG and perform subsequent analyses. Resolving partial edges from PAG is an open problem. This is tackled in two ways: (1) Solicit expert advice to orient edges [44], which may be too cumbersome especially when the graph is complex [44]; or (2) Use an complementary structure learning algorithm to suggest correct orientations [55, 94].

In this paper, we to use the latter. We use NOTEARS [102], an unconstrained optimization based structural learning algorithm to learn a directed acyclic graph (DAG) from observational data. NOTEARS is useful for our application because it produces *confidence scores* for each direction of edge orientation. We may this use to orient the partial edges.

To generate a DAG, NOTEARS learns an adjacency matrix from the observational data. NOTEARS initializes an adjacency matrix A with random values (between 0 and 1). Next, it defines a smooth function (h) that measures the “DAG-ness” of the graph such that $h(A) = 0$ iff the adjacency matrix A is acyclic. Finally, it uses an unconstrained optimization to update the values of the adjacency matrix until the adjacency matrix converges. The final adjacency matrix has values that lie between 0 and 1, where 1 indicates a high confidence that there is an edge and 0 otherwise. The directionality of the edge may be determined using the confidence scores, e.g., if $confidence(V_1 \rightarrow V_2) < confidence(V_2 \rightarrow V_1)$, then the edge directionality is $V_2 \rightarrow V_1$ (as seen in the edge between Resource Pressure and swap memory in Fig. 3e).

We obtain a DAG from the NOTEARS adjacency matrix to complement the PAG from FCI. This is done by setting a threshold on the confidence scores in the adjacency matrix. If the confidence of an edge in the adjacency matrix exceeds this threshold, we keep the edge otherwise we remove the edge, e.g., absent (grayed out) edge between GPU growth and swap memory in Fig. 3e. The chosen threshold determines how many edges are retained. If the threshold is too low, then there will be a lot of spurious edges. Alternately, if the threshold is too high, the ADMG will be sparse. For this work, we found empirically that a threshold of 0.75 for the confidence scores represented a reasonable balance.

We compare all the partially directed edges from the FCI’s PAG (Fig. 3d) with their corresponding counterparts from NOTEARS’ DAG (Fig. 3e). We experienced the following three contrasting edges between FCI and NOTEARS:

1. $\circ \rightarrow$ and \rightarrow : In the most common case, we observed that FCI had a partially directed edge and NOTEARS had a directed edge (as seen in the edge between swap memory and latency in Figs. 3d and 3e). Here we kept the directed edge since evidence from NOTEARS provides us the directionality.
2. $\circ \circ \rightarrow$ and \rightarrow : In the second most common case, we observed FCI had an undirected edge and NOTEARS had a directed edge (as seen in the edge between Resource Pressure and swap memory in Figs. 3d and 3e). Here again, we kept the directed edge in accordance with NOTEARS.

³We do not discuss the orientation rules in detail here. These rules have been studied extensively, and we direct interested readers to works of Sprites *et al.* [35, 72, 86] and Colombo *et al.* [19, 20] for a detailed discussion.

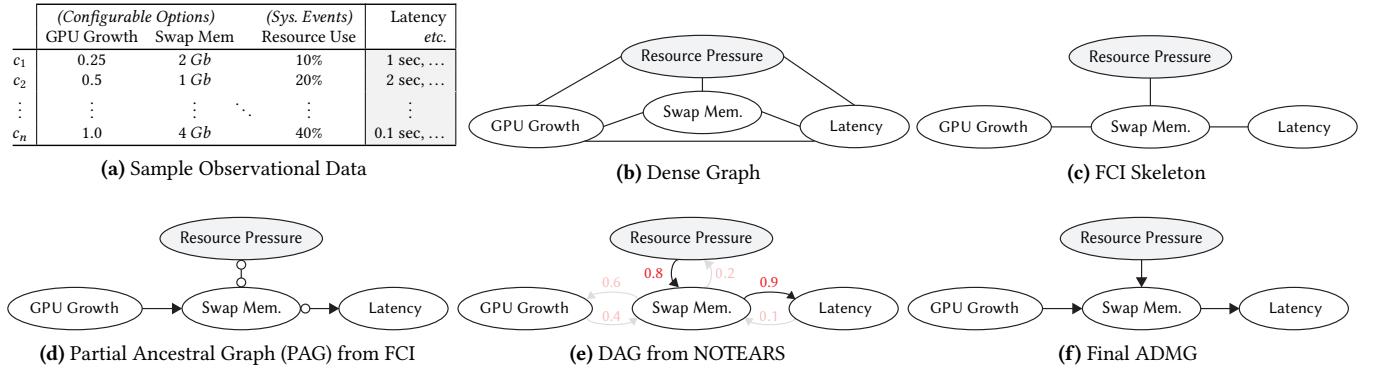


Figure 3. From observational data to a fully connected graph, a skeleton graph, and finally to partial ancestral graph (PAG).

3. \longleftrightarrow and no edge: Rarely, we observed that FCI had an undirected edge and NOTEARS had no edge (as seen in the edge between GPU growth and swap memory in Figs. 3d and 3e). Here, we retain the bidirected edge since there likely is a latent confounder to be accommodated for.

The final causal model would be an ADMG that resembles Fig. 3f. As discussed in §4.4, we update the observational data, and consequently the causal graph and the internal edge directions, periodically with new samples.

4.2 Causal Path Extraction

In this stage, we extract paths from the causal graph (referred to as *causal paths*) and rank them from highest to lowest based on their average causal effect on latency, energy consumption, and heat dissipation (our three non-functional properties). Using path extraction and ranking, we reduce the complex causal graph into a small number of useful causal paths for further analyses. The configurations in this path are more likely to be associated with the root cause of the fault.

Extracting causal paths with backtracking. A causal path is a directed path originating from either the configuration options or the system event 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. If any intermediate node has more than one parent, then we create a path for each parent and continue backtracking on each parent.

Example. From Fig. 3f, we can extract two paths:

- GPU growth \rightarrow swap memory \rightarrow Latency; and
- Resource Pressure \rightarrow swap memory \rightarrow Latency

Note, there may be edges between two non-functional properties, e.g., latency \leftrightarrow energy. However, paths always terminate at a non-functional property, i.e., at latency, energy consumption, and at heat dissipation.

Ranking causal paths. A complex causal graph can result in large number of causal paths. It is intractable to reason

over all possible paths. Therefore, we rank the paths in a 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 as follows:

$$\mathbb{E}[Z | do(X = x)] \quad (2)$$

Eq. (2) 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 $X \rightarrow Z$ (i.e., GPU growth \rightarrow swap memory), we find the average of Eq. (2) over all permissible values the node X (GPU growth) can take, i.e.,

$$ACE(Z, X) = \frac{1}{N} \cdot \sum_{\forall a, b \in X} \mathbb{E}[Z | do(X = b)] - \mathbb{E}[Z | do(X = a)] \quad (3)$$

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. For the entire path, we extend Eq. (3) as follows:

$$Path_{ACE} = \frac{1}{K} \cdot \sum_{\forall X, Z \in path} ACE(Z, X) \quad (4)$$

Eq. (4) represents the average causal effect of the causal path. The configuration options lie in paths with larger P_{ACE} tend to have a greater causal effect on the corresponding non-functional properties in those paths. We select the top K paths with the largest P_{ACE} values, for each non-functional property. In this paper, we use $K=3$, however, this may be modified in our replication package.

4.3 Repairing non-functional faults

In this stage, we use the top K paths 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 high latency, i.e., a latency fault, and has the following questions:

• “**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 in §4.2 to extract the paths from the causal graph and rank the paths based on their average causal effect (i.e., Path_{ACE} from Eq. (4)) on latency. We return the configurations that lie on the top K paths. For example, in Fig. 3f we may return the path (say) GPU growth → swap memory → Latency and the configuration options GPU growth and swap memory both being probable root causes.

• “**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 in order that the new latency is better (low latency) than the fault (high latency). For example, we consider the causal path GPU growth ↔ swap memory → 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:

$$\Pr(Y_{\text{repair}}^{\text{LOW}} | \neg \text{repair}, Y_{\neg \text{repair}}^{\text{HIGH}}) \quad (5)$$

Eq. (5) measures the probability of “fixing” the latency fault with a “repair” ($Y_{\text{repair}}^{\text{LOW}}$) given that with no repair we observed the fault ($Y_{\neg \text{repair}}^{\text{HIGH}}$). 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, i.e.,

$$\mathcal{R} \equiv \bigcup_{\forall x \in \text{GPU growth}, z \in \text{swap memory}, \dots} \{ \text{GPU growth} = x, \text{swap memory} = z, \dots \} \quad (6)$$

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 given by:

$$\text{ITE}(r) = \Pr(Y_r^{\text{LOW}} | \neg r, Y_{\neg r}^{\text{HIGH}}) - \Pr(Y_r^{\text{HIGH}} | \neg r, Y_{\neg r}^{\text{LOW}}) \quad (7)$$

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 . If this difference is positive, then the repair has a higher chance of fixing the fault. In contrast, if the difference is negative then that repair will likely worsen the latency. To find the most useful repair ($\mathcal{R}_{\text{best}}$), we find the repair with the largest (positive) ITE, i.e.,

$$\mathcal{R}_{\text{best}} = \underset{\forall r \in \mathcal{R}}{\operatorname{argmax}} [\text{ITE}(r)] \quad (8)$$

This provides the developer with a possible repair for the configuration options that can fix the latency fault.

Remarks. The ITE computation of Eq. (7) occurs *only* on the observational data. Therefore we may generate any number of repairs and reason about them without having to deploy those intervention and measuring their performance in real-world. This offers significant monetary and runtime benefits.

4.4 Incremental Learning

In this stage, we generate a new configuration using the recommended repairs from Eq. (8). We reconfigured system with the new configuration and we observe the system behavior. If the new configuration addresses 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 Eq. (8) derived from the current version of the causal graph. The more accurate the causal graph, the more accurate Eq. (8) will be [19, 20, 35, 72, 86]. 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 causal effect will become more accurate. We terminate the incremental learning once we achieve the desired performance.

We do not offer theoretical upper bounds for the convergence of the causal model. We shall address this in our future work. That said, our empirical evidence shows that our models converges after at most 50 additional samples.

4.5 Limitations of CADET

No tool is perfect, CADET is no exception. The efficacy of CADET depends on several factors such as the representativeness of the observational data or presence of unmeasured confounders that negatively affect the quality of the causal model. An incorrect causal model may lack some crucial connections that may result in detecting spurious root causes or recommending incorrect repairs. One promising direction to address this problem would be to refine the causal model with developer feedback. Alternatively, we could transfer a causal model certified by experts for one domain to another equivalent domain. We shall explore these in future work.

CADET uses a query engine to translate common user queries into counterfactual statements. The query translation framework described in this section may be extended to answer more complex queries involving several more configurations and non-functional properties by reformulating the counterfactual expressions in Eq. (5). As part of our future, we hope to develop a formal translation framework (e.g., in the form of a DSL) that can convert any user provided query into a meaningful counterfactual statement.

5 Case Study: Latency Fault in TX2

This section revisits the real-world latency fault previously discussed in §2.1. 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,

Configuration Options	CADET	SMAC	BugDoc	Forum	ACE [†]
					3%
	✓	✓	✓	✓	
CPU Cores	✓	✓	✓	✓	3%
CPU Frequency	✓	✓	✓	✓	6%
EMC Frequency	✓	✓	✓	✓	13%
GPU Frequency	✓	✓	✓	✓	22%
Scheduler Policy	.	✓	✓	.	.
Sched rt runtime	.	.	✓	.	.
Sched child runs	.	.	✓	.	.
Dirty bg. Ratio
Dirty Ratio	.	.	✓	.	.
Drop Caches	.	✓	✓	.	.
CUDA_STATIC	✓	✓	✓	✓	55%
Cache Pressure
Swappiness	.	✓	✓	.	1%
Latency (TX2 frames/sec)	26	24	20	23	
Latency Gain (over TX1)	53%	42%	21%	39%	
Latency Gain (over default)	6.5x	6x	5x	5.75x	
Resolution time	24 mins	4 hrs	3.5 hrs	2 days	

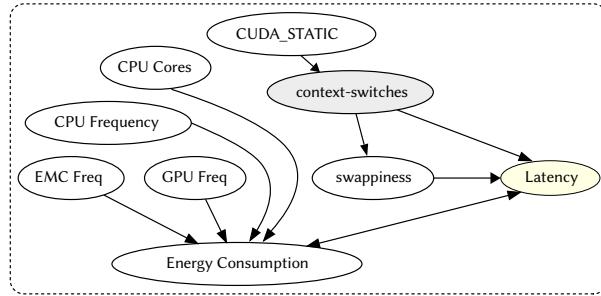


Figure 4. Using CADET on the real-world example from §2.1. CADET is better and faster than other methods.

we use SMAC (an optimization approach) and BugDoc (an ML-based diagnosis tool). We evaluate the methods against the recommendations by the domain experts on the forum.

Findings. 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 6.5x better than the fault). This exceeds the developers' initial expectation of 30 – 40% (or 22 – 24 FPS).
- Using configuration optimization (SMAC), we auto-tune the system to find a near-optimal configuration. This (near-optimal) configuration had a latency of 24 FPS (which was 42% better than TX1 and 6x better than the fault). While SMAC also meets the developer's expectations, it performed slightly worse than CADET, and took 4 hours (6x longer than CADET). Further, the near-optimal configuration found by SMAC, change several unrelated configurations which were not recommended by the experts (depicted by ✓ in Fig. 4).

- BUGDOC (ML-based approach) has the least improvement compared to other approaches (21% improvement over TX1) while taking 3.5 hours (mostly spent on collecting training samples to train internal the decision tree.). BUGDOC failed to meet the developer's expectation. As with SMAC, BugDoc also 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 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 in a timely manner. 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 24% 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.

6 Experimental Setup

6.1 Study subjects

Hardware Systems. This study uses three NVIDIA Jetson Platforms: TX1, TX2, and XAVIER [39, 64]. Each platform has different hardware specifications, e.g., different CPU micro-architectures (ARM Carmel, Cortex), GPU micro-architectures (Maxwell, Pascal, Volta), energy requirements (5W–30W), and thermal management (passive, fans, etc) [1].

Software systems. We deploy five software systems on each NVIDIA Jetson Platform: (1) CNN based Image recognition with Xception to classify 5000 images from the CIFAR10 dataset [18, 57]; (2) BERT (a transformer based model) to perform sentiment analysis on 10000 reviews from the IMDb dataset [26, 61]; (3) DeepSpeech an RNN based voice recognition on 5sec long audio files [43]; (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 resolution 1920 x 1080.

Configuration Options. This work uses 28 configuration options that includes 10 software configurations, 8 OS/Kernel configurations, and 10 hardware configurations (c.f. Fig. 4 for

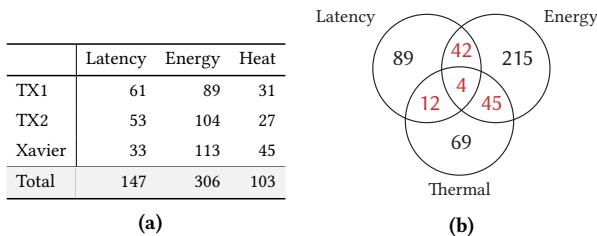


Figure 5. Summary of non-functional faults in JETSON FAULTS dataset. Fig. (a) Distribution across hardware, Fig. (b) Distribution across fault type (red indicate cases with multiple fault types).

some sample configurations). We also record the status of 17 non-intervenable system events. We choose these configurations and system events based on NVIDIA’s configuration guides/tutorials and other related work [39].

6.2 Benchmark Dataset

We curate a non-functional faults dataset, called the JETSON FAULTS dataset, for each of the software and hardware system used in our study. This dataset and the data collection scripts are available in our [replication package](#).

Data Collection. We exhaustively set each of configuration option to all permitted values. Note, for configuration options that were continuous we choose 10 equally spaced values between the minimum and maximum permissible values, e.g., for GPU growth, we vary the value between 0% and 100% in steps of 10%. Next we measure the latency, energy consumption, and heat dissipation for every configuration. We repeat each measurement 5 times and record the average to handle system noise and other variabilities [48].

Labeling misconfigurations. By definition, non-functional faults have latency, energy consumption, and heat dissipation that take tail values [37, 54], 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 summarize the distribution of non-functional faults in each hardware across all software in Fig. 5.

Ground Truth. We create a *ground-truth data* by inspecting each configuration labeled faulty in Fig. 5 and identifying their root-causes manually. We also manually reconfigure each faulty configuration to achieve the best possible latency, energy consumption, and/or heat dissipation thereby dispensing with the non-functional fault. For each reconfiguration, we record the changed configuration options and the values; these are used as a reference for evaluation.

6.3 Evaluation Metrics

Relevance scores. We evaluate the predicted root-causes in terms of (a) the true positive rate (recall), (b) the true discovery rate (precision), and (c) accuracy (jaccard similarity). We prefer high accuracy, precision, and recall. To compute these

metrics, we compare the set of configuration options identified by CADET to be the root cause with the true root-cause from the ground truth (§6.2).

Repair quality. To assess the quality of fixes, we measure the percentage improvement (gain %) after applying the recommended repairs using Δ_{gain} defined as: $\Delta_{gain} = \frac{NFP_{FAULT} - NFP_{NOFAULT}}{NFP_{FAULT}} \times 100$. Here NFP_{FAULT} is the value of the faulty non-functional property (latency, etc.) and $NFP_{NOFAULT}$ is the the value of the faulty non-functional property after applying the repairs recommended by CADET. The larger the Δ_{gain} , the better the quality of the recommended fix.

7 Evaluation

7.1 CADET vs. Model Based Diagnostics

Machine learning approaches are commonly used in model based fault diagnostics [11, 60, 85, 100]. These models learn the correlation between the configuration options and the non-functional properties (e.g, latency, etc). ML-based methods require a diverse and representative set examples to train their models. These models are then extrapolated to diagnose and fix faults. However, learning only correlation can be misleading since it cannot incorporate the intrinsic complex causal structure of the underlying configuration space. In contrast, CADET relies on causal inference (instead of correlation) to model the configuration space thereby overcoming the limitations with current ML-models.

This section compares CADET with four state-of-the-art ML-based methods for fault diagnostics, namely:

- **DELTADEBUGGING** [11]: A debugging technique that minimizes the difference between a pair of configurations, where one configuration causes a fault while the other does not;
- **CBI** [85]: A correlation based feature selection algorithm for identifying and fixing fault inducing configurations;
- **BugDoc** [60]: A Debugging Decision Tree based approach to automatically infer the root causes and derive succinct explanations of misconfiguration induced failures;
- **ENCORE** [100]: An templated based technique that learns correlational information about misconfigurations from a given set of sample configurations.

Note. All approaches require some initial observational data to operate. For CADET we begin with 25 initial samples to let CADET incrementally generate, evaluate, and update the causal model with candidate repairs. Other methods require a large and diverse pool of observational data for training. However, collecting observational data is expensive and time consuming. Therefore, we use a budget of 3 hours to generate random configuration samples to train ML-based methods.

Diagnostics of Single-Objective Faults. In this section, 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

Table 1. Efficiency of CADET compared to other approaches. Cells highlighted in **blue** indicate improvement over faults and **red** indicate deterioration. Overall, CADET achieves better performance overall and is much faster.

(a) Single objective performance fault in latency, energy consumption, or thermal dissipation.

		TX2	Accuracy				Precision				Recall				Gain				Time [†]									
			CADET		CBI	δ-DEBUG	EnCORE	BugDoc	CADET		CBI	δ-DEBUG	EnCORE	BugDoc	CADET		CBI	δ-DEBUG	EnCORE	BugDoc								
			Latency	Energy	Image	84	66	65	68	71	84	67	61	63	67	80	64	68	69	62	81	48	42	57	59	0.6	4	
XAVIER	Energy	NLP	76	65	60	66	66	77	57	55	61	73	66	74	68	67	65	74	54	59	62	58	0.2	4				
		Speech	75	64	63	63	72	71	58	69	61	68	79	73	61	63	69	77	59	53	55	66	0.7	4				
		x264	76	67	60	61	70	74	69	58	65	66	77	64	67	23	72	23	9	12	8	11	1.2	4				
		SQLite	84	65	68	65	70	70	61	62	70	70	84	69	69	64	69	19	13	11	12	8	0.5	4				
		Image	74	63	55	63	64	73	56	58	66	65	80	69	55	63	68	83	59	50	35	51	0.2	4				
TX1	Thermal	NLP	77	60	63	66	64	71	62	64	64	65	66	61	54	63	66	63	49	36	49	53	0.4	4				
		Speech	73	66	65	61	71	75	55	59	54	68	79	53	52	59	71	82	64	48	65	63	1.1	4				
		x264	74	62	57	59	67	81	63	53	61	66	77	67	53	54	72	26	13	11	16	16	0.1	4				
		SQLite	80	53	62	66	71	80	52	66	64	72	84	53	67	65	69	21	16	10	14	15	0.5	4				
		Image	69	63	57	64	65	75	56	56	60	66	68	70	58	64	62	3	3	2	2	2	0.7	4				
Energy + Latency	Latency	NLP	71	62	61	61	62	72	56	59	56	61	72	65	62	67	62	5	4	1	2	4	0.4	4				
		Speech	71	61	64	62	67	71	58	59	54	68	69	67	66	68	67	3	4	2	2	2	1.1	4				
		x264	74	65	57	64	65	74	62	54	55	65	74	66	63	68	67	7	3	2	2	3	0.2	4				
		SQLite	66	64	54	64	65	74	60	54	55	65	65	62	62	68	62	6	2	2	2	3	0.9	4				
		Image	77	54	55	65	73	53	54	62	80	59	59	62	83	53	61	65	70	38	46	44	3	0	0	0	0.6	4
All Three	Energy	NLP	70	51	56	65	71	42	56	63	66	59	62	65	68	53	59	61	60	41	27	48	5	0	0	1	0.2	4
		Speech	74	50	61	65	68	44	53	62	79	51	59	64	82	55	55	62	66	43	43	41	4	-2	-1	-1	0.7	4
		x264	83	54	55	66	81	50	54	57	77	63	62	61	15	2	4	6	13	4	6	4	2	-3	-1	0	1.2	4
		SQLite	84	51	58	68	80	43	55	62	84	57	63	65	11	5	4	5	14	3	8	8	2	-2	0	-1	0.5	4
		Image	76	57	48	66	68	61	57	61	81	53	46	70	62	33	30	42	52	23	18	24	4	1	0	0	0.1	4
	Latency	x264	80	59	47	54	76	61	56	63	81	56	46	51	12	2	1	2	15	4	2	4	4	1	0	1	0.1	4
		SQLite	73	56	51	53	68	59	56	60	78	54	45	51	12	1	1	4	8	4	2	5	1	1	-1	-1	0.1	4

(b) Multi-objective non-functional faults in *Energy, Latency*; and *Energy, Latency, Heat*.

		TX2	Accuracy				Precision				Recall				Gain (Latency)				Gain (Energy)				Gain (Heat)				Time [†]	
			CADET		CBI	EnCORE	BugDoc	CADET		CBI	EnCORE	BugDoc	CADET		CBI	EnCORE	BugDoc	CADET		CBI	EnCORE	BugDoc	CADET		CBI	EnCORE	BugDoc	
			Latency	Energy	Image	77	54	55	65	73	53	54	62	80	59	59	62	83	53	61	65	70	38	46	44	3	0	0
XAVIER	Energy	NLP	70	51	56	65	71	42	56	63	66	59	62	65	68	53	59	61	60	41	27	48	5	0	0	1	0.2	4
		Speech	74	50	61	65	68	44	53	62	79	51	59	64	82	55	55	62	66	43	43	41	4	-2	-1	-1	0.7	4
		x264	83	54	55	66	81	50	54	57	77	63	62	61	15	2	4	6	13	4	6	4	2	-3	-1	0	1.2	4
		SQLite	84	51	58	68	80	43	55	62	84	57	63	65	11	5	4	5	14	3	8	8	2	-2	0	-1	0.5	4
		Image	76	57	48	66	68	61	57	61	81	53	46	70	62	33	30	42	52	23	18	24	4	1	0	0	0.1	4
TX1	Energy	NLP	80	59	47	54	76	61	56	63	81	56	46	51	12	2	1	2	15	4	2	4	4	1	0	1	0.1	4
		Speech	73	56	51	53	68	59	56	60	78	54	45	51	12	1	1	4	8	4	2	5	1	1	-1	-1	0.1	4
		x264	73	56	51	53	68	59	56	60	78	54	45	51	12	1	1	4	8	4	2	5	1	1	-1	-1	0.1	4
		SQLite	73	56	51	53	68	59	56	60	78	54	45	51	12	1	1	4	8	4	2	5	1	1	-1	-1	0.1	4
		Image	76	57	48	66	68	61	57	61	81	53	46	70	62	33	30	42	52	23	18	24	4	1	0	0	0.1	4

[†] Wallclock time in hours

in XAVIER, and heat dissipation faults in TX1. Our findings generalize over other hardware.

Results. Table 1a summarizes the effectiveness of CADET over other ML-based fault diagnosis approaches. We observe the following:

- **Accuracy, precision, and recall.** CADET significantly outperforms ML-based methods in all cases. For example, in image recognition with Xception on TX2, CADET achieves 13% more accuracy, 18% higher recall, and 17% larger Precision compared to BugDoc (best among the rest). We observe similar trends in energy faults, i.e., CADET outperforms other methods in all cases.

- **Gain.** 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 1a, the last two columns indicate the time taken (in hours) by each approach to diagnose the root-cause. For all methods,

we set a maximum budget of 4 hours. We find that, while other approaches use the entire budget to diagnose and resolve the faults, CADET can do so significantly faster, 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 expend most of their allocated 4 hour budget on gathering these training samples. In contrast, CADET starts with only 25 samples and uses incremental learning (§4.4) to judiciously update the causal graph with new configurations until a repair has been found. This drastically reduces the inference time.

Diagnosing Multi-Objective Faults. This section focuses on multi-objective faults where misconfigurations affect multiple non-functional properties simultaneously, i.e., any combination of latency, energy, and heat dissipation (see Fig. 5b).

We report our results for (*Latency, Energy*) faults and (*Latency, Energy, thermal*) faults. The findings in TX2 is representative of multi-objective non-functional faults all across other software and hardware combinations. Complete results are available in the [replication package](#).

Results. From Table 1b, we observe the following:

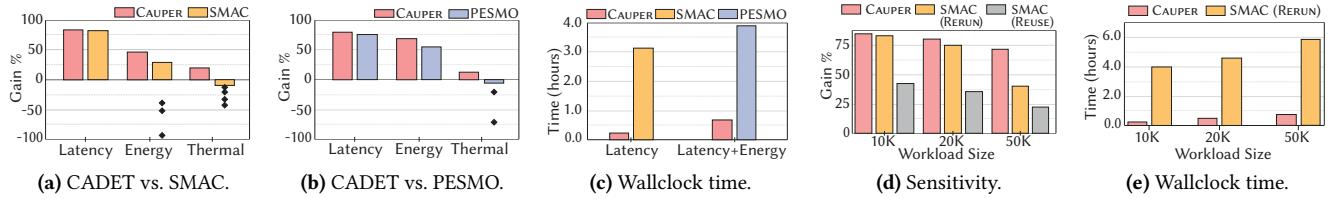


Figure 6. CADET vs. optimization with SMAC and PESMO. CADET performs as well as (or better) than both approaches with considerably shorter inference time. CADET does not cause performance degradation in any non-functional property.

- **Accuracy, Precision, and Recall.** In all cases, we find that CADET could diagnose and fix misconfigurations more effectively compared to other approaches. On average, accuracy, precision, and recall of CADET was 15% higher.
- **Gain.** Except for CADET, all other methods lead to a degradation in one or more non-functional properties (highlighted in red in Table 1b). This is a limitation of ML-models; some configurations options have *multiple effects*, e.g., increasing core frequency reduces latency, but it increases energy consumption and also increases heat dissipation (more energy and cooling is required to accommodate higher clock speeds). ML-models cannot model such nuances, while the causal model used in CADET can.
- **Wallclock time.** CADET is significantly faster than ML-based approaches. ML-based methods have an initial overhead of gathering training samples which depletes the allocated budget of 4 hours. Since CADET uses incremental learning with only 25 initial samples, it is much faster. On average, CADET is 7× faster than ML-based methods.

Discussion. Tables 1a and 1b shows that, DNN applications had the most improvement 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, precision, recall, and *gain* were lower than those for latency and energy consumption faults. There believe there are two reason for this: (1) The workloads exercised in this 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 were relatively lower.

7.2 CADET vs. Search-Based Optimization

Search based methods use approaches such as Bayesian optimization [81, 84] to explore the relationships between configuration options and corresponding non-functional properties. Unlike machine learning models, optimization methods start by building a surrogate model with a small initial sample set. Then they actively search the configuration space to decide which configuration to sample next to improve the

existing surrogate model. Due to the high-dimensionality of the configuration space, search based methods spend a long time exploring sub-optimal configuration during the search process. This quickly depletes available resources.

Although Bayesian optimization has shown much recent promise [25, 46, 65], they are not viable for diagnostics because our objective is not to find an optimal configuration but to fix an already encountered fault. We explore the challenges (and perils) of using optimization for fault diagnostics by comparing CADET with two optimization approaches:

- SMAC [47]: A sequential model based optimization technique to auto-tune software systems;
- PESMO [45]: A multi-objective bayesian optimization to find an near-optimum configuration.

How effective is optimization? Fig. 6a compares CADET with SMAC for fixing latency faults in TX2 (for Image recognition). The fixes recommended by CADET were observed to have better latency gains than the near-optimal configuration discovered by SMAC.

Most notably, in numerous cases, SMAC generated “near-optimal” configurations which in fact lead to significant deterioration in other non-functional properties such as energy consumption and heat dissipation (denoted by ♦ in Fig. 6a). Out of 21 latency faults in image recognition on TX2, we found 7 cases (33%) where the near-optimal configuration caused a significant deterioration in energy consumption and/or heat dissipation. In one case the optimal configuration increased the energy consumption by 96%!

CADET never deteriorates any non-functional property. This is a desirable benefit of using the causal graphs. For example, if CADET notices that a configuration option (say) GPU frequency affects both latency and energy consumption, CADET would not recommend changing GPU frequency to the extent where it consumes too much energy. By design CADET would explore the possibility of changing other configuration options to improve latency. Optimization approaches are not equipped to handle this.

To overcome the above problem, we use a multi-objective optimization such as PESMO [45] to find a configuration that optimizes two or more non-functional properties simultaneously, e.g., optimize energy and latency (without changing heat dissipation). We extend CADET to accommodate this by slightly altering the counterfactual query (c.f. §4.3).

Fig. 6b compares CADET with PESMO for fixing energy and latency faults in TX2 (for Image recognition) while keeping heat dissipation unchanged. Similar to the previous case, we find that CADET performs better than PESMO, i.e., CADET recommends fixes misconfigurations that result in better gain than the near-optimal configuration found by PESMO. Note that, while PESMO did not find a configuration that caused high energy consumption, it still suffered with deterioration in heat dissipation in two cases.

Analysis time of optimization. The analysis time of all approaches is reported in Fig. 6c. Optimization methods run until they converge, however it is a common practice to assign a maximum budget (we use 4 hours) to limit inference time and cost. CADET has significantly lesser inference time compared to the other methods. While SMAC takes slightly more than 3 hours to find a near-optimum in TX2 (image recognition), CADET resolved the faults in 20 minutes on average (9x faster). Likewise, PESMO takes close 4 hours compared to 45 minutes with CADET (5x faster).

- *Why CADET works faster?* CADET uses causal models to infer candidate fixes using only the available observational data. During incremental learning, CADET judiciously evaluates the most promising fixes until the fault is resolved. This limits the number of times the system is reconfiguration and profiled. In contrast, optimization techniques sample the configuration space exhaustively to find an optimal configuration. Every sampled configuration has is deployed and profiled. Optimization techniques sample several sub-optimal configurations in their search process and they are all evaluated. The search space becomes exponentially harder to explore when performing multi-objective optimization.

Sensitivity to external changes in the software. Configuration optimization methods are very sensitive to minor changes to the software stack. To demonstrate this, we use three larger additional image recognition workloads: 10K, 20K, and 50K test images (previous experiments used 5K test images). We evaluate two variants of SMAC: (1) SMAC (REUSE) where we *reuse* the near-optimum found with a 5K tests image on the larger workloads; and (2) SMAC (RERUN) where we *rerun* SMAC afresh on each workload.

CADET performs better than the two variants of SMAC (c.f. Fig. 6d). SMAC (REUSE) performs the worst when the workload changes. With 10K images, reusing the near-optimal configuration from 5K images results in latency gain of 39%, compared to 79% with CADET. With a workload size of 50K images, SMAC (REUSE) only achieves a latency improvement of 24% over the faulty configurations where as CADET finds a fix that improves latency by 72%. SMAC (RERUN) performs better than SMAC (REUSE) but worse than CADET.

In Fig. 6e we report the inference times for SMAC (RERUN) and CADET. In keeping with the previous results, SMAC (RERUN) takes significantly longer than CADET, i.e., SMAC (RERUN) exceeds the 4 hour budget in all workloads whereas CADET takes at most 30 minutes for the largest workload to

diagnose and fix the latency faults. We have to rerun SMAC every time the workload changes, and this limits its practical usability. In contrast, CADET incrementally updates the internal causal model with new samples from the larger workload to learn new relationships from the larger workload. Therefore, it is less sensitive and much faster.

8 Related Work

Performance debugging. Researchers have developed sampling [62], static [41, 59, 67, 69], and end-to-end techniques to detect performance bugs and optimize performance bottlenecks [97, 98] in configurable systems. Several debugging approaches have been proposed for detecting structured workflows using noisy logs [92], anomaly diagnosis [51, 90], data driven approaches [22, 24, 99], explanation tables [28], query based diagnosis [91], statistical debugging and association rule mining based approaches [42, 56, 60, 63, 85, 100], and similarity analysis [71]. These methods are costly as they require running expensive experiments for acquiring reliable performance measurements.

Causal inference. Causal inference had previously been applied to analyse program failures using run-time control and data flow behaviour [12, 13]. Some techniques have also used statistical causal inference on observational data for software fault localization [14, 23, 31, 52]. These approaches are suitable to detect faults in a program but cannot be applied to detect non-functional faults where such workflow is absent. Recently, counterfactual reasoning and causal effect estimation methods have been successfully developed in the context of advertisement recommendation systems [15] and determination of efficient resource sharing strategy for cloud computing [34]. The technique in [30] detects root causes of a software bug using interventions utilizing temporal predicates. In the context of performance debugging, these temporal relationships are not valid; hence, cannot be used to detect root causes. Therefore, we develop CADET to identify the root causes of non-functional faults in configurable systems.

9 Conclusion

Modern computer systems are highly-configurable with thousands interacting configurations with a 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, 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 diagnose the root cause of non-functional faults and recommend high-quality repairs to mitigate these faults.

References

- [1] [n.d.]. NVIDIA Jetson Documentation. [https://docs.nvidia.com/jetson/.](https://docs.nvidia.com/jetson/)
- [2] 2017. NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge. In JetsonHacks Blog: <https://www.jetsonhacks.com/2017/03/25/nvmodel-nvidia-jetson-tx2-development-kit/>.
- [3] 2017. PCIE x4, only 658MB/s - Jetson & Embedded Systems / Jetson TX2. In NVIDIA Developer Forums: <https://forums.developer.nvidia.com/t/pcie-x4-only-658mb-s/55357>.
- [4] 2017. Slow Image Classification with Tensorflow on TX2. In NVIDIA developer forums: <https://forums.developer.nvidia.com/t/54307>.
- [5] 2018. Question about thermal management. In NVIDIA developer forums: <https://forums.developer.nvidia.com/t/59855>.
- [6] 2019. 99% of misconfiguration incidents in the cloud go unnoticed. In Help Net Security: <https://www.helpnetsecurity.com/2019/09/25/cloud-misconfiguration-incidents/>.
- [7] 2019. Jetson Nano not responding or responding very slow over SSH. In NVIDIA Developer Forums: <https://forums.developer.nvidia.com/t/jetson-nano-not-responding-or-responding-very-slow-over-ssh/79530/3>.
- [8] 2020. CUDA performance issue on TX2. In NVIDIA developer forums: <https://forums.developer.nvidia.com/t/50477>.
- [9] 2020. General Performance Problems. In NVIDIA developer forums: <https://forums.developer.nvidia.com/t/111704>.
- [10] 2020. High CPU usage on Jetson TX2 with GigE fully loaded. In NVIDIA developer forums: <https://forums.developer.nvidia.com/t/124381>.
- [11] Cyrille Artho. 2011. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer* 13, 3 (2011), 223–246.
- [12] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 307–320.
- [13] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis.. In *OSDI*, Vol. 10. 1–14.
- [14] George K Baah, Andy Podgurski, and Mary Jean Harrold. 2010. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*. 73–84.
- [15] Léon Bottou, Jonas Peters, Joaquin Quiñonero-Candela, Denis X Charles, D Max Chickering, Elon Portugaly, Dipankar Ray, Patrice Simard, and Ed Snelson. 2013. Counterfactual reasoning and learning systems: The example of computational advertising. *The Journal of Machine Learning Research* 14, 1 (2013), 3207–3260.
- [16] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. 2003. *Computer systems: a programmer's perspective*. Vol. 2. Prentice Hall Upper Saddle River.
- [17] Chia-Chang Chiu and Yi-Sheng Chueh. 2012. Method and computer system for thermal throttling protection. US Patent 8,301,873.
- [18] François Fleuret. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1251–1258.
- [19] Diego Colombo and Marloes H Maathuis. 2014. Order-independent constraint-based causal structure learning. *The Journal of Machine Learning Research* 15, 1 (2014), 3741–3782.
- [20] Diego Colombo, Marloes H Maathuis, Markus Kalisch, and Thomas S Richardson. 2012. Learning high-dimensional directed acyclic graphs with latent and selection variables. *The Annals of Statistics* (2012), 294–321.
- [21] Lynne M Connnelly. 2016. Fisher's exact test. *Medsurg Nursing* 25, 1 (2016), 58–60.
- [22] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. {REPT}: Reverse Debugging of Failures in Deployed Software. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 17–32.
- [23] Charlie Curtsinger and Emery D Berger. 2015. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 184–197.
- [24] Charles M Curtsinger. 2016. Effective Performance Analysis and Debugging. (2016).
- [25] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. 2017. BOAT: Building auto-tuners with structured Bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*. 479–488.
- [26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [27] Michael Ditty, Ashish Karandikar, and David Reed. 2018. Nvidia's xavier SoC. In *Hot Chips: A Symposium on High Performance Chips*.
- [28] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. 2014. Interpretable and informative explanations of outcomes. *Proceedings of the VLDB Endowment* 8, 1 (2014), 61–72.
- [29] Robin J Evans and Thomas S Richardson. 2014. Markovian acyclic directed mixed graphs for discrete data. *The Annals of Statistics* (2014), 1452–1482.
- [30] Anna Fariha, Suman Nath, and Alexandra Meliou. 2020. Causality-Guided Adaptive Interventional Debugging. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 431–446.
- [31] Farid Feyzi and Saeed Parsa. 2019. Inforence: effective fault localization based on information-theoretic analysis and statistical causal inference. *Frontiers of Computer Science* 13, 4 (2019), 735–759.
- [32] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2015. Automated multi-objective control for self-adaptive software design. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 13–24.
- [33] Dustin Franklin. 2017. NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge. In NVIDIA Developer Blog: <https://developer.nvidia.com/blog/jetson-tx2-delivers-twice-intelligence-edge/>.
- [34] Philipp Geiger, Lucian Carata, and Bernhard Schölkopf. 2016. Causal models for debugging and control in cloud computing. *arXiv preprint arXiv 1603* (2016).
- [35] Clark Glymour, Kun Zhang, and Peter Spirtes. 2019. Review of causal discovery methods based on graphical models. *Frontiers in genetics* 10 (2019), 524.
- [36] Jonathan Greig. 2020. Cloud misconfigurations cost companies nearly \$5 trillion. In TechRepublic: <https://www.techrepublic.com/article/cloud-misconfigurations-cost-companies-nearly-5-trillion/>.
- [37] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. 2018. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)* 14, 3 (2018), 1–26.
- [38] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 301–311.
- [39] Hassan Halawa, Hazem A. Abdelhafez, Andrew Boktor, and Matei Ripeanu. 2017. NVIDIA jetson platform characterization. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics) 10417 LNCS* (2017), 92–105. https://doi.org/10.1007/978-3-319-64203-1_7
- [40] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empirical Software Engineering* 24, 2 (2019), 674–717.
- [41] Xue Han and Tingting Yu. 2016. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software*

- Engineering and Measurement.* 1–10.
- [42] Xue Han, Tingting Yu, and David Lo. 2018. PerfLearner: learning from bug reports to understand and generate performance test frames. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 17–28.
 - [43] Awini Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).
 - [44] Yang-Bo He and Zhi Geng. 2008. Active learning of causal networks with intervention experiments and optimal designs. *Journal of Machine Learning Research* 9, Nov (2008), 2523–2547.
 - [45] Daniel Hernández-Lobato, Jose Hernandez-Lobato, Amar Shah, and Ryan Adams. 2016. Predictive entropy search for multi-objective bayesian optimization. In *International Conference on Machine Learning*. 1492–1501.
 - [46] Chin-Jung Hsu, Vivek Nair, Vincent W Freeh, and Tim Menzies. 2018. Arrow: Low-level augmented bayesian optimization for finding the best cloud vm. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 660–670.
 - [47] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*. Springer, 507–523.
 - [48] Md Shahriar Iqbal, Lars Kotthoff, and Pooyan Jamshidi. 2019. Transfer Learning for Performance Modeling of Deep Neural Network Systems. In *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*. 43–46.
 - [49] Md Shahriar Iqbal, Jianhai Su, Lars Kotthoff, and Pooyan Jamshidi. 2020. FlexiBO: Cost-Aware Multi-Objective Optimization of Deep Neural Networks. *arXiv preprint arXiv:2001.06588* (2020).
 - [50] Pooyan Jamshidi and Giuliano Casale. 2016. An uncertainty-aware approach to optimal configuration of stream processing systems. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 39–48.
 - [51] Tong Jia, Pengfei Chen, Lin Yang, Ying Li, Fanjing Meng, and Jingmin Xu. [n.d.]. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In *ICWS 2017*. 25–32.
 - [52] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal Testing: Understanding Defects' Root Causes. In *Proceedings of the 2020 International Conference on Software Engineering*.
 - [53] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.
 - [54] Martin Kleppmann. 2017. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. "O'Reilly Media, Inc".
 - [55] Rex B Kline. 2015. *Principles and practice of structural equation modeling*. Guilford publications.
 - [56] Rahul Krishna, Tim Menzies, and Lucas Layman. 2017. Less is more: Minimizing code reorganization using XTREE. *Information and Software Technology* 88 (2017), 53–66.
 - [57] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. [n.d.]. CIFAR-10 (Canadian Institute for Advanced Research). ([n. d.]). <http://www.cs.toronto.edu/~kriz/cifar.html>
 - [58] Philipp Leitner and Jürgen Cito. 2016. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology (TOIT)* 16, 3 (2016), 1–23.
 - [59] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. 2020. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
 - [60] Raoni Lourenço, Juliana Freire, and Dennis Shasha. 2020. BugDoc: A System for Debugging Computational Pipelines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2733–2736.
 - [61] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Portland, Oregon, USA, 142–150. <http://www.aclweb.org/anthology/P11-1015>
 - [62] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 643–654.
 - [63] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. 2020. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 435–448.
 - [64] Sparsh Mittal. 2019. A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *J. Syst. Archit.* 97, January (2019), 428–442. <https://doi.org/10.1016/j.sysarc.2019.01.011>
 - [65] Takashi Miyazaki, Issei Sato, and Nobuyuki Shimizu. 2018. Bayesian optimization of hpc systems for energy efficiency. In *International Conference on High Performance Computing*. Springer, 44–62.
 - [66] I Molyneaux. 2009. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*.[SI]" O'Reilly Media.
 - [67] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An empirical study of real-world variability bugs detected by variability-oblivious tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 50–61.
 - [68] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Faster discovery of faster system configurations with spectral learning. *Automated Software Engineering* 25, 2 (2018), 247–277.
 - [69] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE, 902–912.
 - [70] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (MSR)*. IEEE, 237–246.
 - [71] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 562–571.
 - [72] Juan Miguel Ogarrio, Peter Spirtes, and Joe Ramsey. 2016. A hybrid causal search algorithm for latent variable models. In *Conference on Probabilistic Graphical Models*. 368–379.
 - [73] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. 2018. Architectural principles for cloud software. *ACM Transactions on Internet Technology (TOIT)* 18, 2 (2018), 1–23.
 - [74] Judea Pearl. 2009. *Causality*. Cambridge university press.
 - [75] Judea Pearl. 2011. The algorithmization of counterfactuals. *Annals of Mathematics and Artificial Intelligence* 61, 1 (2011), 29.
 - [76] Judea Pearl et al. 2000. Models, reasoning and inference. *Cambridge, UK: Cambridge University Press* (2000).
 - [77] Judea Pearl and Dana Mackenzie. 2018. *The book of why: the new science of cause and effect*. Basic Books.
 - [78] C Mylara Reddy and N Nalini. 2016. Fault Tolerant Cloud Software Systems Using Software Configurations. In *2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. IEEE, 61–65.
 - [79] Thomas Richardson, Peter Spirtes, et al. 2002. Ancestral graph Markov models. *The Annals of Statistics* 30, 4 (2002), 962–1030.

- [80] Ana B Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. 2020. TANDEM: A Taxonomy and a Dataset of Real-World Performance Bugs. *IEEE Access* 8 (2020), 107214–107228.
- [81] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.
- [82] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 284–294.
- [83] Norbert Siegmund, Sergiy S Kolesnikov, Christian Kästner, Sven Apel, Don Batory, Marko Rosenmüller, and Gunter Saake. 2012. Predicting performance via automated feature-interaction detection. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 167–177.
- [84] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*. 2951–2959.
- [85] Linhai Song and Shan Lu. 2014. Statistical debugging for real-world performance problems. *ACM SIGPLAN Notices* 49, 10 (2014), 561–578.
- [86] Peter Spirtes, Clark N Glymour, Richard Scheines, and David Heckerman. 2000. *Causation, prediction, and search*. MIT press.
- [87] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 271–280.
- [88] Sokratis Tsakiltsidis, Andriy Miranskyy, and Elie Mazzawi. 2016. On automatic detection of performance bugs. In *2016 IEEE international symposium on software reliability engineering workshops (ISSREW)*. IEEE, 132–139.
- [89] Pavel Valov, Jean-Christophe Petkovich, Jianmei Guo, Sebastian Fischmeister, and Krzysztof Czarnecki. 2017. Transferring performance prediction models across different hardware platforms. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 39–50.
- [90] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. 2015. Data x-ray: A diagnostic tool for data errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1231–1245.
- [91] Xiaolan Wang, Alexandra Meliou, and Eugene Wu. 2017. QFix: Diagnosing errors through query histories. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1369–1384.
- [92] Fei Wu, Pranay Anchuri, and Zhenhui Li. 2017. Structural event detection from log messages. In *23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1175–1184.
- [93] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep parameter optimisation. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. 1375–1382.
- [94] Shen Xinpeng, Ma Sisi, Vemuri Prashanthi, Gyorgy Simon, Michael W Weiner, Paul Aisen, Ronald Petersen, Clifford R Jack, Andrew J Saykin, William Jagust, et al. 2020. Challenges and Opportunities with Causal Discovery Algorithms: Application to Alzheimer’s Pathophysiology. *Scientific Reports (Nature Publisher Group)* 10, 1 (2020).
- [95] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 307–319.
- [96] Nezih Yigitbasi, Theodore L Willke, Guangdeng Liao, and Dick Epema. 2013. Towards machine learning-based auto-tuning of mapreduce. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 11–20.
- [97] Tingting Yu and Michael Pradel. 2016. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *25th International Symposium on Software Testing and Analysis*. 389–400.
- [98] Tingting Yu and Michael Pradel. 2018. Pinpointing and repairing performance bottlenecks in concurrent programs. *Empirical Software Engineering* 23, 5 (2018), 3034–3071.
- [99] Xiao Yu. 2018. Understanding and Debugging Complex Software Systems: A Data-Driven Perspective. (2018).
- [100] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niuy Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 687–700.
- [101] Yi Zhang, Jianmei Guo, Eric Blais, and Krzysztof Czarnecki. 2015. Performance prediction of configurable software systems by fourier learning (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 365–373.
- [102] Xun Zheng, Bryon Aragam, Pradeep K Ravikumar, and Eric P Xing. 2018. DAGs with NO TEARS: Continuous optimization for structure learning. In *Advances in Neural Information Processing Systems*. 9472–9483.